sun microsystems

System Services Overview

Part Number: 800-3846-10
Revision A of 27 March, 1990

Trademarks

SunOS™, Sun Workstation®, as well as the word "Sun" followed by a numerical suffix, are trademarks of Sun Microsystems, Incorporated.

UNIX® and UNIX System V® are trademarks of Bell Laboratories.

PostScript™ is a trademark of Adobe Systems Inc.

All other products or services mentioned in this document are identified by the trademarks or service marks of their respective companies or organizations.

Legal Notice to Users

Yellow Pages™ is a registered trademark in the United Kingdom, of British Telecommunications plc., and may also be a trademark of various telephone companies around the world. Sun will be revising future versions of software and documentation to remove references to the term "Yellow Pages."

The Sun Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

This product is protected by one or more of the following U.S. patents: 4,777,485 4,688,190 4,527,232 4,745,407 4,679,014 4,435,792 4,719,569 4,550,368 in addition to foreign patents and applications pending.

This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following individuals and institutions for their role in its development: The Regents of the University of California, the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California, and Other Contributors.

Contents

Tables

Figures

1

Introduction

1.1. Overview

Release 4.1 of the SunOS operating system (hereafter referred to as ''Release 4.1,'' or ''4.1'') is derived from Berkeley Standard Distribution (BSD) release 4.3, which in turn, was derived form Version 7 of the UNIX operating system developed at Bell Laboratories. 4.1 also incorporates numerous features from UNIX System V Release 3, including library routines that are compliant with the SVID, Issue 2, STREAMS-based communication facilities, RFS, and System V interprocess communication facilities.

System services are typically made available to an executing program (process) by means of library routines (function calls). Services provided by the system kernel are described in the *Kernel Interface* chapter. Network-based services and networking concepts are introduced in the *Networking Overview* chapter. For a detailed description of the various system abstractions in Release 4.1, refer to `Intro`(2) and `Intro`(3) in the *SunOS Reference Manual.*

This manual also describes the architecture of the virtual memory system, in *The Virtual Memory System.* Programming security features are outlined in *Programmer's Guide to Security Features.*

1.2. Compatibility and Conformance

An important feature of the SunOS operating system is its compatibility and conformance with various emerging standards for the UNIX operating system. This manual also describes how Release 4.1 complies with these various standards.

2

The Virtual Memory System

Release 4.1 of the SunOS operating system provides a virtual-memory system with a rich set of memory-management facilities. These facilities, in turn, form a basis for providing system services such as shared libraries.

Process address spaces are composed of a vector of memory pages, each of which can be independently mapped and manipulated. Typically, the system presents mappings that simulate the traditional UNIX process memory environment, but other views of memory are useful as well.

These memory-management facilities:

- Unify the system's operations on memory.
- Provide a set of kernel mechanisms powerful and general enough to support the implementation of fundamental system services without special-purpose kernel support.
- Maintain consistency with the existing environment, in particular using the file system as the name space for named virtual-memory objects.

2.1. Virtual Memory, Address Spaces and Mapping

The system's *virtual memory* (VM) consists of all available physical memory resources. Examples include local and remote file systems, processor primary memory, swap space and other random-access devices. Named objects in the virtual memory are referenced though the file system. However, not all file system objects are in the virtual memory; devices that the operating system cannot treat as storage, such as terminal and network device files, are not in the virtual memory. Some virtual memory objects, such as private process memory and System V shared memory segments (refer to *Programming Utilities and Libraries*), are not named.

A process's *address space* is defined by mappings onto objects in the system's virtual memory (usually files). Each mapping is constrained to be sized and aligned with the page boundaries of the system on which the process is executing. Each page may be mapped (or not) independently. Only process addresses that are mapped to some system object are valid, for there is no memory associated with processes themselves—all memory is represented by objects in the system's virtual memory.

Each object in the virtual memory has an *object address space* defined by some physical storage. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory's associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store.

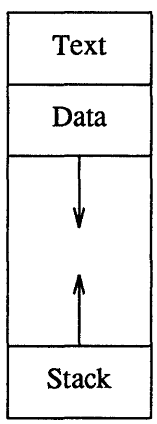
A given process page may map to only one object, although a given object address may be the subject of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made is not affected by the mere *existence* of the mapping. Thus, it cannot, in general, be expected that an object has an ''awareness'' of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a ''page'' is not a property of the object. Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through `read()` and `write()`. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (single-step updates rather than the `read()`, modify buffer, `write()` cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

Address Space Layout

Traditionally, the address space of a process has consisted of exactly three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process's stack. Text is read-only and shared, while the data and stack segments are private to the process. as follows:

Figure 2-1 *Traditional UNIX System Address-Space Layout*

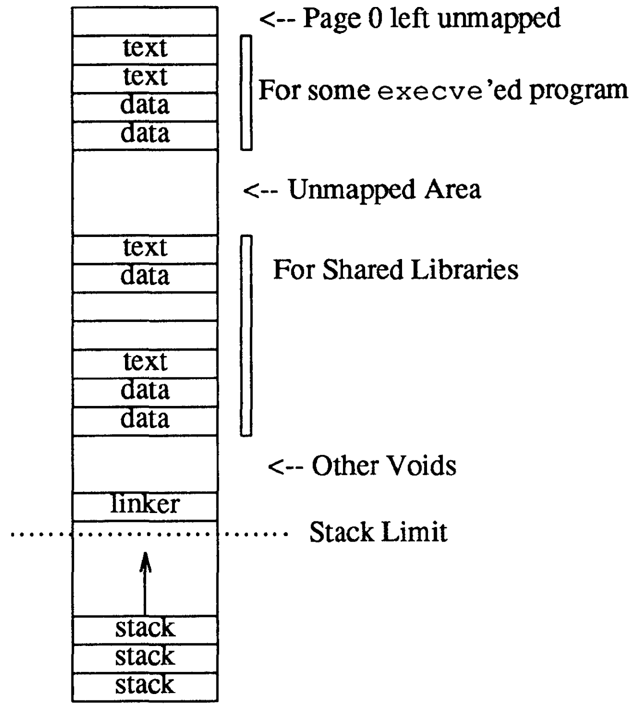


Under Release 4.1, a process's address space is simply a vector of pages, and the division between different address-space segments is not so clear-cut. Process text and data spaces are simply groups of pages.[1] There are often multiple text and data "segments", some belonging to specific programs and some belonging

[1] For compatibility purposes, the system maintains address ranges that "should" belong to such segments to support operations such as extending or contracting the data segment's "break". These are initialized when a program is initiated with `execve()`.

to code running in shared libraries. An illustration of one possible layout of an address space is:

Figure 2-2 *Address-space Layout*



Release 4.1 system processes still uses text, data, and stack segments, but these are better thought of as constructs provided by the programming environment rather than the operating system. As such, it is possible to construct processes that have multiple segments of each "type," or of types of arbitrary semantic value — no longer are programs restricted to being built only from objects the system was capable of representing directly. For instance, a process's address space may contain multiple text and data segments, some belonging to specific programs and some shared among multiple programs. Text segments from shared libraries, for example, typically appear in the address spaces of many processes. A process's address space is simply a vector of pages, and there is no necessary division between different address-space segments. Process text and data spaces are simply groups of pages mapped in ways appropriate to the function they provide the program.

A process's address space is usually sparsely populated, with data and text pages intermingled. The precise mechanics of the management of stack space is machine-dependent, although by convention, page 0 is not used. Process address spaces are often constructed through dynamic linking when a program is `exec`'ed. Operations such as `exec()` and dynamic linking build upon the mapping operations described previously. Dynamic linking is described further in *Programming Utilities and Libraries*.

While the system may have multiple areas that can be considered "data" segments, for programming convenience the system maintains operations to operate on an area of storage associated with a process's initial "heap storage area." A process can manipulate this area by calling `brk()` and `sbrk()`:

```
caddr_t brk(addr)
caddr_t addr;

caddr_t sbrk(incr);
int incr;
```

`brk()` sets the system's idea of the lowest data segment location not used by the caller to `addr` (rounded up to the next multiple of the system's page size).

`sbrk()`, the alternate function, adds `incr` bytes to the caller's data space and returns a pointer to the start of the new data area.

Shared Memory

Memory sharing between processes (or even between two areas of the same process) occurs whenever mappings are establish that reference the same memory object. This can occur when two processes map common addresses of a single file, or when a parent and child share a `MAP_SHARED` mapping across a `fork()`.

This memory sharing is an *implicit* form of Interprocess Communication (IPC), which is turns out to be a highly efficient method for communicating information between processes. Within this framework, the general form of establishing common memory for mapping into multiple processes for purposes IPC is to create a file. However, for compatibility purposes, Release 4.1 also provides the standard System V shared memory segments, along with messages and semaphores. These facilities are described in *Programming Utilities and Libraries*.

2.2. Networking, Heterogeneity and Coherence

The VM is designed to fit well with the operating system's heterogeneous environment, an environment that makes extensive use of networking to access file systems which can now be regarded as part of the system's virtual memory.

Networks are not constrained to consist of similar hardware or to be based upon a common operating system; in fact, the opposite is encouraged, for such constraints create serious barriers to accommodating heterogeneity. While a given set of processes may *apply* a set of mechanisms to establish and maintain the properties of various system objects—properties such as page sizes and the ability of objects to synchronize their own use—a given operating system should not *impose* such mechanisms on the rest of the network.

As it stands, the access-method view of virtual memory maintains the potential for a given object (say a text file) to be mapped by the operating system's memory-management facilities, and also by systems like PC-DOS, for which virtual memory and storage management techniques such as paging are totally foreign. Such systems can continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The unacceptable alternative would be to prohibit access to the object by less capable systems.

Another consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In both of these cases, the object is being shared, and thus the applications must use some synchronization mechanism to guarantee the coherence of their transactions with it. The scope and nature of the synchronization mechanism is best left to the application to decide. For example, file access on systems that do not support virtual memory access methods must be indirect, by way of `read()` and `write()`. Applications sharing files on such systems must coordinate their access using semaphores, file locking or some application-specific protocols. What is required in an environment where mapping replaces `read()` and `write()` as the access method is an operation, such as `fsync()`, that supports atomic update operations.

The nature and scope of synchronization over shared objects is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, the needs of applications are met without erecting barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the "operating system."

2.3. Memory Management Interfaces

The applications programmer gains access to the facilities of the VM system through several sets of system calls. This section summarizes these calls, and provides examples of their use. For details, see the *SunOS Reference Manual*.

Creating and Using Mappings

```
caddr_t mmap(addr, len, prot, flags, fd, off)
caddr_t addr;
size_t len;
int prot, flags, fd;
off_t off;
```

`mmap()` establishes a mapping between a process's address space and an object in the system's virtual memory. It is the system's most fundamental function for defining the contents of an address space — all other system functions that contribute to the definition of an address space are built from `mmap()`. The format of an `mmap()` call is:

```
paddr = mmap(addr, len, prot, flags, fd, off);
```

`mmap()` establishes a mapping from the process's address space at an address `paddr` for `len` bytes to the object specified by `fd` at offset `off` for `len` bytes. The value returned by `mmap()` is an implementation-dependent function of the parameter `addr` and the setting of the `MAP_FIXED` bit of `flags`, as described below. A successful call to `mmap()` returns `paddr` as its result. The address range `[paddr, paddr + len)` must be valid for the address space of the process and the range `[off, off + len)` must be valid for the virtual memory object. (The notation `[start, end)` refers to the interval from `start` to `end`, including `start` but not including `end`.) The mapping established by `mmap()` replaces any previous mappings for the process's pages in the range `[paddr, paddr + len)`.

The parameter prot determines whether read, execute, write or some combination of accesses are permitted to the pages being mapped. Specify permissions by an OR of the flags values PROT_READ, PROT_EXECUTE, and PROT_WRITE. A write access must fail if PROT_WRITE has not been set, though the behavior of the write can be influenced by setting MAP_PRIVATE in the flags parameter, as described below.

The flags parameter provides other information about the handling of mapped pages:

- MAP_SHARED and MAP_PRIVATE specify the mapping type, and one of them must be specified. The mapping type describes the disposition of store operations made by *this* process into the address range defined by the mapping operation. If MAP_SHARED is specified, write references will modify the mapped object. No further operations on the object are necessary to effect a change — the act of storing into a MAP_SHARED mapping is equivalent to doing a write() system call.

On the other hand, if MAP_PRIVATE is specified, an initial write reference to a page in the mapped area will create a copy of that page and redirect the initial and successive write references to that copy. This operation is sometimes referred to as *copy-on-write* and occurs invisibly to the process causing the store. Only pages actually modified have copies made in this manner. MAP_PRIVATE mappings are used by system functions such as exec(2) when mapping files containing programs for execution. This permits operations by programs such as debuggers to modify the ''text'' (code) of the program without affecting the file from which the program is obtained. The private copy is not created until the first write; until then, other users who have the object mapped MAP_SHARED can change the object. That is, if one user has an object mapped MAP_PRIVATE and another user has the same object mapped MAP_SHARED, and the MAP_SHARED user changes the object before the MAP_PRIVATE user does the first write, then the changes appear in the MAP_PRIVATE user's copy that the system makes on the first write. If an application desires such isolation, it should use read to make a copy of the data it wishes to keep isolated.

The mapping type is retained across a fork(). The mapping type only affects the disposition of stores by the calling process—there is no isolation from changes made by other processes. If an application desires such isolation, it should use read() to make a copy of the data it wishes to keep isolated.

- MAP_FIXED informs the system that the value returned by mmap() must be addr, exactly. The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of system resources. When MAP_FIXED is not set, the system uses addr as a hint to arrive at paddr. The paddr so chosen is an area of the address space that the system deems suitable for a mapping of len bytes to the specified object. An addr value of zero grants the system complete freedom in selecting paddr, subject to constraints described below. A non-zero value of addr is taken as a suggestion of a process address near which the mapping should be placed. When the system selects a value for paddr, it never places a mapping at address 0, nor replaces any extant mapping, nor maps into areas considered part of the potential data or stack "segments." The system strives to choose alignments for mappings that maximize the performance of the its hardware resources.

The file descriptor used in a `mmap()` call need not be kept open after the mapping is established. If it is closed, the mapping will remain until such time as it is replaced by another call to `mmap()` that explicitly specifies the addresses occupied by this mapping; or until the mapping is removed either by process termination or a call to `munmap()`. Although the mapping endures independently of the existence of a file descriptor, changes to the file can influence accesses to the mapped area, even if they do not affect the mapping itself. For instance, should a file be shortened by a call to `truncate()`, such that the mapping now "overhangs" the end of the file, then accesses to that area of the file that "does not exist" will result in `SIGBUS` signals. It is possible to create the mapping in the first place such that it "overhangs" the end of the file — the only requirement when creating a mapping is that the addresses, lengths, and offsets specified in the operation be *possible* (i.e., within the range permitted for the object in question), not that they exist at the time the mapping is created (or subsequently.)

Similarly, if a program accesses an address in a manner inconsistently with how it has been mapped (for instance, a store operation into a mapping that was established with only `PROT_READ` access), then a `SIGSEGV` signal will result. `SIGSEGV` signals will also result on any attempt to reference an address not defined by any mapping.

In general, if a program makes a reference to an address that is inconsistent with the mapping (or lack of a mapping) established at that address, the system will respond with a `SIGSEGV` violation. However, if a program makes a reference to an address consistent with how the address is mapped, but that address does not evaluate *at the time of the access* to allocated storage in the object being mapped, then the system will respond with a `SIGBUS` violation. In this manner a program (or user) can distinguish between whether it is the mapping or the object that is inconsistent with the access, and take appropriate remedial action.

Using `mmap()` to access system memory objects can simplify programs in a variety of ways. Keeping in mind that `mmap()` can really be viewed as just a means to access memory objects, it is possible to program using `mmap()` in many cases where you might program with `read()` or `write()`. However, it is important to realize that `mmap()` can only be used to gain access to *memory* objects — those objects that can be thought of as randomly accessible storage. Thus, terminals and network connections can not be accessed with `mmap()` because they are not "memory." Magnetic tapes, even though they are memory devices, can not be accessed with `mmap()` because storage locations on the tape can only be addressed sequentially. Some examples of situations that *can* be thought of as candidates for use of `mmap()` over more traditional methods of file access include:

- Random access operations — either map the entire file into memory or, if the address space can not accommodate the file or if the file size is variable, create "windows" of mappings to the object.

- Efficiency — even in situations where access is sequential, if the object being accessed can be accessed using `mmap()`, an efficiency gain may be obtained by avoiding the copying operations inherent in accesses via `read()` or `write()`. For even greater efficiency, you can use `madvise()` to set the `MADV_SEQUENTIAL` flag, in which case the system will free each page after it is passed.

- Structured storage — if the storage being accessed is collected as tables or data structures, algorithms can be more conveniently written if access to the file is treated just as though the tables were in memory. Previously, programs could not simply make storage or table alterations in memory and save them for access in subsequent runs, however when the addresses of the table are defined by mappings to a file then changes to the storage *are* changes to the file, and are thus automatically recorded in it.

  Scattered storage — if a program requires scattered regions of storage, such as multiple heaps or stack areas, such areas can be defined by mapping operations during program operation. However, this method is not portable to systems using the traditional UNIX address-space layout.

The remainder of this section will illustrate some other concepts surrounding mapping creation and use.

Mapping `/dev/zero` gives the calling program a block of zero-filled virtual memory of the size specified in the call to `mmap()`. `/dev/zero` is a special device, that responds to `read()` as an infinite source of bytes with the value 0, but when mapped creates an unnamed object to back the mapped region of memory. The following code fragment demonstrates a use of this to create a block of scratch storage in a program, at an address of the system's choosing.

```
/*
 * Function to allocate a block of zeroed storage.  Parameter
 * is the number of bytes desired.  The storage is mapped as
 * MAP_SHARED, so that if a fork() occurs, the child process
 * will be able to access and modify the storage.  If we wished
 * to cause the child's modifications (as well as those by the
 * parent) to be invisible to the ancestry of processes, we
 * would use MAP_PRIVATE.
 */
caddr_t get_zero_storage(len)
int len;
{
    int fd;
    caddr_t result;

    if ((fd = open("/dev/zero", O_RDWR)) == -1)
        return ((caddr_t)-1);
    result = mmap(0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    (void) close(fd);
    return (result);
}
```

As written, this function permits a hierarchy of processes to use the area of allocated storage as a region of communication for *implicit* Interprocess Communication. As noted earlier, System V IPC facilities can be used to accomplish the same purpose without requiring that the processes be in a parent-child hierarchy.

In some cases, devices or files are *only* useful when accessed by way of mapping. An example of this are frame buffer devices used to support bit-mapped displays, where display management algorithms function best if they can operate randomly on the addresses of the display directly.

Finally, it is important to remember that mappings can be operated upon at the granularity of a single page. Even though a mapping operation may define multiple pages of an address space, there is *absolutely no restriction* that subsequent operations on those addresses must operate on the same number of pages. For instance, an `mmap()` operation defining 10 pages of an address space may be followed by subsequent `munmap()` (see below) operations that remove every other page from the address space, leaving 5 mapped pages each followed by an unmapped page. Those unmapped pages may subsequently be mapped to different locations in the same or different objects, or the whole range of pages (or any partition, superset, or subset of the pages) used in other `mmap()` or other memory management operations. Further, it must be noted that any mapping operation that operates on more than a single page can ''partially succeed'' in that some parts of the address range can be affected even though the call returns a failure. Thus, an `mmap()` operation that replaces another mapping, if it fails, may have deleted the previous mapping and failed to replace it. Similarly, other operations (unless specifically stated otherwise) may process some pages in the range successfully before operating on a page where the operation fails.

Removing Mappings

```
int munmap(addr, len)
caddr_t addr;
size_t len;
```

`munmap()` removes all mappings in the range `[addr, addr + len)` from the address space of the calling process. It is not an error to remove mappings from addresses that do not have them, and *any* mapping, no matter how it was established, can be removed with `munmap()`. `munmap()` does not in any way affect the objects that were mapped at those addresses.

Cache Control

The memory management system in Release 4.1 can be thought of as a form of ''cache management,'' in which a processor's primary memory is used as a cache for pages from objects from the system's virtual memory. Thus, there are a number of operations that control or interrogate the status of this ''cache,'' as described in this section.

```
int mincore(addr, len, vec)
caddr_t addr;
size_t len;
char *vec;
```

`mincore()` determines the residency of the memory pages in the address space covered by mappings in the range `[addr, addr + len)`. Using the ''cache concept'' described earlier, this function can be viewed as an operation that interrogates the status of the cache, and returns an indication of what is currently resident in the cache. The status is returned as a char-per-page in the character array referenced by *vec (which the system assumes to be large enough to encompass all the pages in the address range). Each character contains either a ''1'' (indicating that the page is resident in the system's primary storage), or a ''0'' (indicating that the page is not resident in primary storage.) Other bits in the character are reserved for possible future expansion — therefore programs testing residency should test only the least significant bit of each character.

```
int mlock(addr, len)
caddr_t addr;
size_t len;

int munlock(addr, len)
caddr_t addr;
size_t len;
```

mlock() causes the pages referenced by the mapping in the range [addr, addr + len) to be locked in physical memory. References to those pages (even through other mappings in this or other processes) will not result in page faults that require an I/O operation to obtain the data needed to satisfy the reference. Because this operation ties up physical system resources, and has the potential to disrupt normal system operation, use of this facility is restricted to the super-user. The system will not permit more than a configuration-dependent limit of pages to be locked in memory simultaneously, the call to mlock() will fail if this limit is exceeded.

munlock() releases the locks on physical pages. Note that if multiple mlock() calls are made through the same mapping, only a single munlock() call will be required to release the locks (in other words, locks on a given mapping do not nest.) However, if different mappings to the same pages are processed with mlock(), then the pages will not be unlocked until the locks on all the mappings are released.

Locks are also released when a mapping is removed, either through being replaced with an mmap() operation or removed explicitly with munmap(). A lock will be transferred between pages on the ''copy-on-write'' event associated with a MAP_PRIVATE mapping, thus locks on an address range that includes MAP_PRIVATE mappings will be retained transparently along with the copy-on-write redirection (see mmap() above for a discussion of this redirection.)

```
int mlockall(flags)
int flags;

int
munlockall()
```

mlockall() and munlockall() are similar in purpose and restriction to mlock() and munlock(), except that they operate on entire address spaces. mlockall() accepts a flags argument that influences whether the lock is to affect everything currently in the address space, everything that will be added in the future, or both. The flags are built as a bit-field of values from the set:

```
MCL_CURRENT                  Current mappings
MCL_FUTURE                   Future mappings
```

munlockall() removes all locks on all pages in the address space, whether established by mlock() or mlockall().

```
int msync(addr, len, flags)
caddr_t addr;
size_t len;
int flags;
```

msync() supports applications that require assertions about the integrity of data in the storage backing their mapping, either for correctness or for coherent communications in a distributed environment. msync() causes all modified copies of pages over the range [addr, addr + len) to be flushed to the objects mapped by those addresses. In the cache analogy discussed previously, msync() is the cache "write-back," or flush, operation. It is similar in purpose to the fsync() operation for files.

msync() optionally invalidates such cache entries so that further references to the pages cause the system to obtain them from their permanent storage locations.

The flags argument provides a bit-field of values that influences the behavior of msync(). The bit names and their interpretations are:

MS_SYNC	Synchronized write
MS_ASYNC	Return immediately
MS_INVALIDATE	Invalidate caches

MS_SYNC causes msync() to return only after all I/O operations are complete. MS_ASYNC causes msync() to return immediately once all I/O operations are scheduled. MS_INVALIDATE causes all cached copies of data from mapped objects to be invalidated, requiring them to be re-obtained from the object's storage upon the next reference.

Other Mapping Functions

```
int
getpagesize()
```

getpagesize() returns the system-dependent size of a memory page. For portability, applications should not embed any constants specifying the size of a page, and instead should make use of getpagesize() to obtain that information. Note that it is not unusual for page sizes to vary even among implementations of the same instruction set, increasing the importance of using this function for portability.

```
int mprotect(addr, len, prot)
caddr_t addr;
size_t len;
int prot;
```

`mprotect()` has the effect of assigning protection `prot` to all pages in the range `[addr, addr + len)`. The protection assigned can not exceed the permissions allowed on the underlying object. For instance, a read-only mapping to a file that was opened for read-only access can not be set to be writable with `mprotect()` (unless the mapping is of the `MAP_PRIVATE` type, in which case the write access is permitted since the writes will modify copies of pages from the object, and not the object itself.)

```
int munmap(addr, len)
caddr_t addr;
size_t len;
```

`munmap()` has the effect of removing all pages in the range `[addr, addr + len)` from the address space of the calling process.

```
int
getpagesize()
```

`getpagesize()` returns the system-dependent size of a memory page.

```
int mincore(addr, len, vec)
caddr_t addr;
size_t len;
char *vec;
```

`mincore()` determines the residency of the memory pages in the address space covered by mappings in the range `[addr, addr + len)`. The status is returned as a char-per-page in the character array referenced by `*vec` (which the system assumes to be large enough to encompass all the pages in the address range).

3

Kernel Interface

3.1. Processes and Protection

Host and Process Identifiers

Each host system has associated with it a 32-bit host ID, and a hostname of up to MAXHOSTNAMELEN characters (as defined in <sys/param.h>). The hostname is accessed and modified with the calls:

```
int getdomainname(name, namelen)
char *name;
int namelen;

int setdomainname(name, namelen)
char *name;
int namelen;

long gethostid()

int gethostname(name, namelen)
char *name;
int namelen;

int sethostname(name, namelen)
char *name;
int namelen;
```

getdomainname() places the name of the domain for the current processor in the string pointed to by the name parameter. name is null-terminated if space allows. setdomainname() sets the name of the current processor's domain to the string pointed to by name.

On each host runs a set of *processes*. Each process is largely independent of other processes, having its own protection domain, address space, timers, and an independent set of references to system or user implemented objects.

Each process in a host is named by an integer called the *process ID* (PID). This number is in the range MAXPID1- (as defined in <sys/param.h>). A process can discover its PID with the getpid() routine:

```
pid_t getpid()
```

On each host this identifier is guaranteed to be unique; in a multi-host environment, the (hostid, PID) pairs are guaranteed unique.

Creating and Terminating Processes

A new process is usually created by copying that mappings that define the address space of a *parent* process, thus making a logical duplicate of the parent. (See the *Virtual Memory System* chapter for a description of mapping).

```
pid_t fork()
```

The `fork()` call returns twice, once in the parent process, where the PID is the process identifier of the child, and once in the child process where the PID is 0.

Since `execve()` (see below) specifies `MAP_PRIVATE` on all the mappings it performs, parent and child effectively have copy-on-write access to a single set of objects. Any `MAP_SHARED` mappings in the parent are also `MAP_SHARED` in the child, providing the opportunity for both parent and child to operate on a common object. The parent-child relationship induces a hierarchical structure on the set of processes in the system.

A process may terminate by executing an `exit()` call:

```
int exit(status)
int status;
```

returning 8 bits of exit status to its parent.

When a child process exits or terminates abnormally, the parent process receives information about any event which caused termination of the child process. A second call provides a non-blocking interface and may also be used to retrieve information about resources consumed by the process during its lifetime.

```
#include <sys/wait.h>
#include <sys/resource.h>

int wait(statusp)
union wait *statusp;

int wait3(statusp, options, rusage)
union wait *statusp;
int options;
struct rusage *rusage;
```

The System V-compatible `waitpid`(2V) routine can be used to obtain information about a selected process.

A process can overlay itself with the memory image of another program, passing the newly created process a set of parameters, using the call:

```
int execve(path, argv, envp)
char *path, **argv, **envp;
```

`execve()` specifies `MAP_PRIVATE` on the mappings which overlay the old address space. execve() performs this operation by performing the internal equivalent of an mmap() to the file containing the program. The text and initialized data segments are mapped to the file, and the program's uninitialized data and stack areas are mapped to unnamed objects in the system's virtual memory. The boundaries of the mappings it establishes are recorded as representing the traditional "segments" of a UNIX process's address space.

The text segment is mapped with only PROT_READ and PROT_EXECUTE protections, so that write references to the text produce segmentation violations. The data segment is mapped as writable; however any page of initialized data that does not get written may be shared among all the processes running the program.

The specified name must be a file which is in a format recognized by the system, either a binary executable file or a ASCII file which causes the execution of a specified interpreter program (usually sh(1) or csh(1)) to process its contents.

User and Group Ids

Each process in the system has associated with it two user ID's (UID) a *real* user ID (RUID), and an *effective* user ID (EUID), both non-negative 16 bit integers. (Note: a user may change his EUID, but this does *not* change his RUID). Each process has a real accounting group ID (GID), an effective accounting group ID (EGID), and a set of access group IDs. Group IDs are non-negative 16 bit integers. Each process may be in several different access groups, with the maximum concurrent number of access groups a system compilation parameter, the constant NGROUPS in the file <sys/param.h>, guaranteed to be at least 8.

The real and effective user IDs associated with a process are returned by getuid() and geteuid(), respectively.

```
uid_t getuid()
uid_t geteuid()
```

the real and effective accounting group ID by:

```
gid_t getgid()
gid_t getegid()
```

and the set of access group IDs is placed in the array pointed to by the gidset parameter of getgroups():

```
#include <sys/param.h>

int getgroups(gidsetlen, gidset)
int gidsetlen;
gid_t gidset[];
```

User and group IDs are assigned at login time using the setreuid(), setregid(), and setgroups() calls:

```
int setreuid(ruid, euid)
int ruid, euid;

int setregid(rgid, egid)
int rgid, egid;

#include <sys/param.h>

int setgroups(ngroups, gidset)
int ngroups;
gid_t gidset[];
```

The `setreuid()` call sets both the real and effective user IDs, while the `setregid()` call sets both the real and effective accounting group IDs. Unless the caller is the super-user, the RUID must be equal to either the current real or effective user ID, and RGID equal to either the current real or effective accounting group. The `setgroups()` call is restricted to the super-user.

Process Groups and Controlling Terminals

Each process in the system is also normally associated with a *process group*. The group of processes in a process group is sometimes referred to as a *job* and manipulated by high-level system software (such as the shell). The current process group of a process is returned by the `getpgrp()` call:

```
int getpgrp(pid)
int pid;
```

The process group associated with a process may be changed using `setpgid()`:

```
#include <sys/types.h>

int setpgid (pid, pgid)
pid_t pid, pgid;
```

Newly created processes are assigned process IDs distinct from all processes and process groups, and the same process group as their parent. A normal (unprivileged) process may set its process group equal to its process ID. A privileged process may set the process group of any process to any value.

When a process is in a specific process group it may receive software interrupts affecting the group, causing the group to suspend or resume execution or to be interrupted or terminated. In particular, every system terminal has a process group and only processes which are in the process group of a terminal may read from the terminal, allowing arbitration of terminals among several different jobs. A process can examine the process group of the terminal's foreground process using `tcgetpgrp()`:

```
#include <sys/types.h>

pid_t tcgetpgrp(fd)
int fd;
```

A process may change the process group of any terminal which it can write using: `tcsetpgrp()` call:

```
int tcsetpgrp(fd, pgrp_id)
int fd;
pid_t pgrp_id;
```

The terminal's process group may be set to any value. Thus, more than one terminal may be in a process group.

Controlling Terminal

Each process in the system is usually associated with a *controlling terminal*, accessible through the file `/dev/tty`. A newly created process inherits the controlling terminal of its parent. A process may be in a different process group than its controlling terminal, in which case the process does not receive software interrupts affecting the controlling terminal's process group.

You can arrange for a process to be detached from the controlling terminal using `setsid()`:

```
#include <sys/types.h>

pid_t setsid()
```

Refer to *UNKNOWN TITLE ABBREVIATION: RELEASE* for more information about setting the controlling terminal for a process group.

`tty` Parameters

Certain functions that relate to the state of the terminal device have been repackaged for POSIX conformance and portability. Previous interfaces are still available by way of `ioctl()` requests. The new functions are:

Get/set terminal (line) speeds: `cfgetispeed`(2), `cfsetispeed`(2), `cfgetospeed`(2), and `cfsetospeed`(2).

Line control functions: `tcdrain`(2), `tcflow`(2), and `tcflush`(2).

Get/set attributes (such as line discipline modes): `tcgetattr`(2) and `tcsetattr`(2).

Get/set `tty` process group: `tcgetpgrp`(2), and `tcsetpgrp`(2).

Sessions and Process Groups

Release 4.1 incorporates the concept of a session. A session is a grouping of process groups just as a process group is a grouping of processes. Sessions are closely related to controlling terminals; each controlling terminal belongs to a session. All processes with the same controlling terminal are in the same session. A terminal may be the controlling terminal for at most one session.

`setsid`(2) is a new function that creates a new session with the calling process as the session leader and only member of that session. Note: a session leader may not create a new session by calling `setsid()` a second time. `setsid()` is similar to

```
ioctl(fd, TIOCNOTTY, (char*)0)
```

in that `setsid()` disassociates the calling process from its controlling terminal, if any; the `TIOCNOTTY` `ioctl` has been changed to be a call to `setsid()`.

Process Groups

There is a new version of `setpgrp()` called `setpgid()`; `setpgid()` is POSIX compliant. Release 4.1 supports both, but the meaning of `setpgrp(mypid, 0)` has changed. That particular variation of the system call has been changed to invoke `setsid()`.

`setpgrp()` no longer allows arbitrary values for `pgrp`. A process is only allowed to create a new `pgrp` equal to its PID, or join an existing process group within its session.

In 4.1, a process must be a session leader in order to acquire a controlling terminal. Since `setsid()` is new to 4.1, the system has been modified to call it on the behalf of old binaries. The system makes every effort to arrange that a process is a session leader at the appropriate time such that the process will receive a controlling terminal. For more information refer to *UNKNOWN TITLE ABBREVIATION: RELEASE.*

Deallocating a Controlling Terminal

The following will all result in the deallocation of the process's controlling terminal, provided the process is not a session leader:

```
setpgrp(0, 0);
ioctl(fd, TIOCNOTTY, (char*)0);
setsid();
```

The most portable way to get rid of a controlling terminal is to:

```
 if (fork())
    exit();
(void) setsid();
```

The `fork()` is necessary to make sure the process is not a session leader. For BSD based programs, the `setsid()` call may be safely replaced by a call to `setpgrp(0,0)`. These calls are equivalent on 4.1 and later systems. On earlier systems this will *not* deallocate the controlling terminal; it does modify process state enough that the terminal will be replaced by a different one on the next attempt to open the terminal.

3.2. Signals

The system defines a set of *signals* that may be delivered to a process. Signal delivery resembles the occurrence of a hardware interrupt: the signal is blocked from further occurrence, the current process context is saved, and a new one is built. A process may specify the *handler* to which a signal is delivered, or specify that the signal is to be *blocked* or *ignored*. A process may also specify that a *default* action is to be taken when signals occur.

Some signals will cause a process to exit when they are not caught. This may be accompanied by creation of a `core` image file, containing the current memory image of the process for use in post-mortem debugging. A process may choose to have signals delivered on a special stack, so that sophisticated software stack manipulations are possible.

All signals have the same *priority*. If multiple signals are pending simultaneously, the order in which they are delivered to a process is implementation specific. Signal routines execute with the signal that caused their invocation *blocked*, but other signals may yet occur. Mechanisms are provided whereby critical sections of code may protect themselves against the occurrence of specified signals.

For POSIX compliance, 4.1 includes a new package of signal library routines. The new functions are: `sigaction(2V)` `sigaddset(2V)` `sigdelset(2V)` `sigemptyset(2V)` `sigfillset(2V)` `sigismember(2V)` `sigpending(2V)` `sigprocmask(2V)` and .`sigsuspend(2V)` Another change for POSIX allows the `SIGCONT` signal to be blocked. The effect is that the process is still restarted upon the receipt of a `SIGCONT` signal but the handler is not called until the signal is unblocked.

Signal Types

The signals defined by the system fall into one of five classes: hardware conditions, software conditions, input/output notification, process control, or resource control. The set of signals is defined in the file `<signal.h>`.

Hardware signals are derived from exceptional conditions which may occur during execution. Such signals include `SIGFPE` representing floating point and other arithmetic exceptions, `SIGILL` for illegal instruction execution, `SIGSEGV` for addresses outside the currently assigned area of memory, and `SIGBUS` for accesses that violate memory protection constraints. Other, more cpu-specific hardware signals exist, such as `SIGIOT`, `SIGEMT`, and `SIGTRAP`.

Software signals reflect interrupts generated by user request: `SIGINT` for the normal interrupt signal; `SIGQUIT` for the more powerful quit signal, that normally causes a core image to be generated; `SIGHUP` and `SIGTERM` that cause graceful process termination, either because a user has "hung up", or by user or program request; and `SIGKILL`, a more powerful termination signal which a process cannot catch or ignore. Programs may define their own asynchronous events using `SIGUSR1` and `SIGUSR2`. Other software signals (`SIGALRM`, `SIGVTALRM`, `SIGPROF`) indicate the expiration of interval timers.

A process can request notification via a `SIGIO` signal when input or output is possible on a descriptor, or when a *non-blocking* operation completes. A process may request to receive a `SIGURG` signal when an urgent condition arises.

A process may be *stopped* by a signal sent to it or the members of its process group. The `SIGSTOP` signal is a powerful stop signal, because it cannot be caught. Other stop signals `SIGTSTP`, `SIGTTIN`, and `SIGTTOU` are used when a user request, input request, or output request respectively is the reason for stopping the process. A `SIGCONT` signal is sent to a process when it is continued from a stopped state. Processes may receive notification with a `SIGCHLD` signal when a child process changes state, either by stopping or by terminating.

Exceeding resource limits may cause signals to be generated. `SIGXCPU` occurs when a process nears its CPU time limit and `SIGXFSZ` warns that the limit on file size creation has been reached.

Signal Handlers

A process has a handler associated with each signal. The handler controls the way the signal is delivered. The call:

```
#include <signal.h>

struct sigvec {
    int (*sv_handler)();
    int sv_mask;
    int sv_flags;
};

int sigvec(sig, vec, ovec)
int sig;
struct sigvec *vec, *ovec;
```

assigns interrupt handler address `sv_handler` to signal `sig`. Each handler address specifies either an interrupt routine for the signal, that the signal is to be ignored, or that a default action (usually process termination) is to occur if the signal occurs. The constants `SIG_IGN` and `SIG_DFL` used as values for `sv_handler` cause ignoring or defaulting of a condition.

NOTE *There are two things that must be done to reset a signal handler from within a signal handler. Resetting the routine that catches the signal, which*

```
signal(n, SIG_DFL)
```

does, is only the first. It's also necessary to unblock the blocked signal, which is done with `sigsetmask()` *or* `sigblock()`. *The way to think of signals is as hardware interrupts. Just resetting the vector for the interrupt is not enough, you also have to lower the processor priority level.*

The `sv_mask` and `sv_onstack` values specify the signal mask to be used when the handler is invoked; it implicitly includes the signal which invoked the handler. Signal masks include one bit for each signal; the mask for a signal *signo* is provided by the macro `sigmask(`*signo*`)`, from `<signal.h>`. `sv_flags` specifies whether system calls should be restarted if the signal handler returns and whether the handler should operate on the normal run-time stack or a special signal stack (see below). If `osv` is non-zero, the previous signal vector is returned. It also specifies whether the signal action is to be reset to `SIG_DFL`, and if the signal is to be blocked by setting a bit to the signal mask, when the signal handler is called. This latter behavior is the default; the former is for backward compatibility with the signal mechanisms of some other versions of the UNIX system (V7, BSD4.1, System V, etc.).

When a signal condition arises for a process, the signal is added to a set of signals pending for the process. If the signal is not currently *blocked* by the process then it will be delivered. The process of signal delivery adds the signal to be delivered and those signals specified in the associated signal handler's `sv_mask` to a set of those *masked* for the process, saves the current process context, and places the process in the context of the signal handling routine. The call is arranged so that if the signal handling routine exits normally the signal mask will be restored and the process will resume execution in the original context. If the process wishes to resume in a different context, then it must arrange to restore the signal mask itself.

You can use the `sigpending()` call to inquire about signals that are pending and blocked:

```
#include <signal.h>

int sigpending(set)
sigset_t *set;
```

The mask of *blocked* signals is independent of handlers for delays. It delays the delivery of signals much as a raised hardware interrupt priority level delays hardware interrupts. Preventing an interrupt from occurring by changing the handler is analogous to disabling a device from further interrupts.

The signal handling routine `sv_handler` is called by a C call of the form

```
(*sv_handler)(signo, code, scp, addr)
int signo, code;
struct sigcontext *scp;
char *addr;
```

The `signo` gives the number of the signal that occurred, while `code`, is a parameter of certain signals that provides additional detail. The `scp` parameter is a pointer to a machine-dependent structure containing the information for restoring the context from before the signal. `addr` is additional address information.

Sending Signals

A process can send a signal to another process or group of processes with the calls:

```
int kill(pid, sig)
pid_t pid;
int sig;

int killpg(pgrp, sig)
int pgrp, sig;
```

Unless the process sending the signal is privileged, it must have the same effective user ID as the process receiving the signal.

Signals can also be sent from a terminal device to the process group associated with the terminal. See `kill`(1).

Protecting Critical Sections

To block a section of code against one or more signals, a `sigblock()` call may be used to add a set of signals to the existing mask, returning the old mask:

```
int sigblock(mask)
int mask;
```

The old mask can then be restored later with `sigsetmask()`,

```
int sigsetmask(mask)
int mask;
```

The `sigblock()` call can be used to read the current mask by specifying an empty `mask`.

It is possible to check conditions with some signals blocked, and then to pause waiting for a signal and restoring the mask, by using:

```
int sigpause(sigmask)
int sigmask;
```

Signal Stacks

Applications that maintain complex or fixed size stacks can use the call:

```
struct  sigstack {
    char *ss_sp;
    int  ss_onstack;
};

int sigstack (ss, oss)
struct sigstack *ss, *oss;
```

to provide the system with a stack based at `ss_sp` for delivery of signals. The value `ss_onstack` indicates whether the process is currently on the signal stack, a notion maintained in software by the system.

When a signal is to be delivered, the system checks whether the process is on a signal stack. If not, then the process is switched to the signal stack for delivery, with the return from the signal arranged to restore the previous stack.

If the process wishes to take a non-local exit from the signal routine, or run code from the signal stack that uses a different stack, a `sigstack()` call should be used to reset the signal stack.

3.3. Timers

Real Time

The system's notion of the current Greenwich time and the current time zone is set and returned by the calls:

```
#include <sys/time.h>

int settimeofday(tvp, tzp)
struct timeval *tp;
struct timezone *tzp;

gettimeofday(tp, tzp)
result struct timeval *tp;
result struct timezone *tzp;
```

where the structures are defined in <sys/time.h> as:

```
struct timeval {
    long    tv_sec;      /* seconds since Jan 1, 1970 */
    long    tv_usec;     /* and microseconds */
};

struct timezone {
    int tz_minuteswest; /* of Greenwich */
    int tz_dsttime;     /* type of dst correction to apply */
};
```

The precision of the system clock is hardware dependent. Earlier versions of the UNIX system contained only a 1-second resolution version of this call, which remains as a library routine:

```
#include <sys/time.h>

time_t time(tloc)
time_t *tloc;
```

returning only the `tv_sec` field from the `gettimeofday()` call.

Interval Time

The system provides each process with three interval timers, defined in <sys/time.h>:

```
#define ITIMER_REAL      0    /* real time intervals */
#define ITIMER_VIRTUAL   1    /* virtual time intervals */
#define ITIMER_PROF      2    /* user and system virtual time */
```

The `ITIMER_REAL` timer decrements in real time. It could be used by a library routine to maintain a wakeup service queue. A `SIGALRM` signal is delivered when this timer expires.

The `ITIMER_VIRTUAL` timer decrements in process virtual time. It runs only when the process is executing. A `SIGVTALRM` signal is delivered when it expires.

The `ITIMER_PROF` timer decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by processes to statistically profile their execution. A `SIGPROF` signal is delivered when it expires.

A timer value is defined by the `itimerval` structure:

```
struct itimerval {
    struct  timeval it_interval;    /* timer interval */
    struct  timeval it_value;       /* current value */
};
```

and a timer is set or read by the call:

```
int getitimer(which, value)
int which;
result struct itimerval *value;
int setitimer(which, value, ovalue)
int which;
struct itimerval *value, *ovalue;
```

The third argument to `setitimer()` specifies an optional structure to receive the previous contents of the interval timer. A timer can be disabled by specifying a timer value of 0.

The system rounds argument timer intervals to be not less than the resolution of its clock. This clock resolution can be determined by loading a very small value into a timer and reading the timer back to see what value resulted.

The `alarm()` system call of earlier versions of the UNIX system is provided as a library routine using the `ITIMER_REAL` timer. The process profiling facilities of earlier versions of the UNIX system remain because it is not always possible to guarantee the automatic restart of system calls after receipt of a signal. The `profil()` call arranges for the kernel to begin gathering execution statistics for a process:

```
int profil(buf, bufsize, offset, scale)
char *buf;
int bufsize, offset, scale;
```

This begins sampling of the program counter, with statistics maintained in the user-provided buffer.

3.4. Descriptors

Each process has access to resources through *descriptors*. Each descriptor is a handle allowing the process to reference objects such as files, devices and communications links.

The Reference Table

Rather than allowing processes direct access to descriptors, the system introduces a level of indirection, so that descriptors may be shared between processes. Each process has a *descriptor reference table*, containing pointers to the actual descriptors. The descriptors themselves thus have multiple references, and are reference counted by the system.

Each process has a fixed size descriptor reference table, where the size is returned by the `getdtablesize()` call:

```
int getdtablesize()
```

and guaranteed to be at least 20. The entries in the descriptor reference table are referred to by small integers; for example if there are 20 slots they are numbered 0 to 19.

Descriptor Properties

Each descriptor has a logical set of properties maintained by the system and defined by its *type*. Each type supports a set of operations; some operations, such as reading and writing, are common to several abstractions, while others are unique. Generic operations applying to many of these types are described in *3.7*. Naming contexts, files and directories are described in *3.8*. Section *4.1*. describes communications domains and sockets. Terminals and (structured and unstructured) devices are described in *3.9*.

Managing Descriptor References

A duplicate of a descriptor reference may be made by doing

```
int dup(fd)
int fd;
```

returning a copy of descriptor reference `fd` indistinguishable from the original. The new `fd` chosen by the system will be the smallest unused descriptor reference slot. A copy of a descriptor reference may be made in a specific slot by doing

```
int dup2(old, new)
int old, new;
```

The `dup2()` call causes the system to deallocate the descriptor reference currently occupying slot *new*, if any, replacing it with a reference to the same descriptor as old. This deallocation is also performed by:

```
int close(fd)
int fd;
```

For applications that use a large number of open descriptors, the following routine can be used to count the number of descriptors currently open:

```
#include <sys/stat.h>

static  struct stat fdstat;

int count_open_fds()
{
    int fd;
    int count = 0;
    int max_fds = getdtablesize();
        for (fd = 0; fd < max_fds; fd ++)
            if (fstat(fd, & fdstat) == 0)
                count ++;
    return count;
}
```

Multiplexing Requests

The system provides a standard way to perform synchronous and asynchronous multiplexing of operations.

Note: Operations are said to be multiplexed when they are interleaved in real time on the same device or communications channel. For example, I/O streams A and B are multiplexed if B begins before A is completed.

Synchronous multiplexing is performed by using the `select()` call to examine the state of multiple descriptors simultaneously, and to wait for state changes on those descriptors. Sets of descriptors of interest are specified as bit masks, as follows:

```
#include <sys/types.h>
#include <sys/time.h>

int select (width, readfds, writefds, exceptfds, timeout)
int width;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_ZERO(&fdset)
FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
int fd;
fs_set fdset;
```

The `select()` call examines the descriptors specified by the sets `readfss`, `writefds` and `exceptfds`, replacing the specified bit masks by the subsets that select true for input, output, and exceptional conditions respectively (`width` indicates the number of file descriptors specified by the bit masks). If any descriptors meet the following criteria, then the number of such descriptors is returned, and the bit masks are updated.

- A descriptor selects for input if an input oriented operation such as `read()` or `receive()` is possible, or if a connection request may be accepted (see *Accepting Connections)* in section *4.1.1.*
- A descriptor selects for output if an output oriented operation such as `write()` or `send()` is possible, or if an operation that was "in progress", such as connection establishment, has completed (see section *3.7.3.*
- A descriptor selects for an exceptional condition if a condition that would cause a `SIGURG` signal to be generated exists (see section *3.2.1)* or other device-specific events have occurred.

If none of the specified conditions is true, the operation waits for one of the conditions to arise, blocking at most the amount of time specified by `timeout`. If `timeout` is given as 0, the `select()` waits indefinitely

Options affecting I/O on a descriptor may be read and set by the call:

```
#include <fcntl.h>

int fcntl (des, cmd, arg)
int des, cmd, arg;

/* Interesting values for cmd */
#define F_DUPFD      0    /* Return new descriptor */
#define F_SETFD      1    /* Set close-on-exec flag */
#define F_GETFD      2    /* Set close-on-exec flag */
#define F_SETFL      3    /* Set descriptor options */
#define F_GETFL      4    /* Set descriptor options */
#define F_SETOWN     5    /* Set descriptor owner (pid/pgrp) */
#define F_GETOWN     6    /* Set descriptor owner (pid/pgrp) */
```

The `F_SETFL` `cmd` may be used to set a descriptor in non-blocking I/O mode and/or enable signaling when I/O is possible. `F_SETOWN` *must* be used to specify a process or process group to be signaled when using the latter mode of operation or when urgent indications arise.

Operations on non-blocking descriptors will either complete immediately, note an error `EWOULDBLOCK`, partially complete an input or output operation returning a partial count, or return an error `EINPROGRESS` noting that the requested operation is in progress. A descriptor which has signaling enabled will cause the specified process and/or process group be signaled, with a `SIGIO` for input, output, or in-progress operation complete, or a `SIGURG` for exceptional conditions.

For example, when writing to a terminal using non-blocking output, the system will accept only as much data as there is buffer space for and return; when making a connection on a *socket*, the operation may return indicating that the connection establishment is "in progress". The `select()` facility can be used to determine when further output is possible on the terminal, or when the connection establishment attempt is complete.

3.5. Resource Controls

Process Priorities

The system gives CPU scheduling priority to processes that have not used CPU time recently. This tends to favor interactive processes and processes that execute only for short periods. It is possible to determine the priority currently assigned to a process, process group, or the processes of a specified user, or to alter this priority using the calls:

```
#include <sys/time.h>
#include <sys/resource.h>

#define PRIO_PROCESS    0    /* process */
#define PRIO_PGRP       1    /* process group */
#define PRIO_USER       2    /* user ID */

int getpriority(which, who)
int which, who;

int setpriority(which, who, prio)
int which, who, prio;
```

The value returned by getpriority() is in the range –20 to 20. The default priority is 0; lower priorities cause more favorable execution. The getpriority() call returns the highest priority (lowest numerical value) enjoyed by any of the specified processes. The setpriority() call sets the priorities of all of the specified processes to the specified value. Only the super-user may lower priorities.

Resource Utilization

getrusage() places information about currently consumed resources in a structure defined in <sys/resource.h>:

```
#include <sys/time.h>
#include <sys/resource.h>

#define  RUSAGE_SELF  0         /* usage by this process */
#define  RUSAGE_CHILDREN   -1   /* usage by all children */

getrusage(who, rusage)
int who;
struct rusage *rusage;

struct rusage {
        struct timeval ru_utime;     /* user time used */
        struct timeval ru_stime;     /* system time used */
        long    ru_maxrss;
#define ru_first        ru_ixrss
        /* XXX: In 4.0, all three ru_i?rss fields are combined
         *          and presented in idrss; ixrss and isrss are zero
         */
        long    ru_ixrss;       /* integral shared memory size */
        long    ru_idrss;       /* integral unshared data */
        long    ru_isrss;       /* integral unshared stack */
        long    ru_minflt;      /* page reclaims */
        long    ru_majflt;      /* page faults */
        long    ru_nswap;       /* swaps */
        long    ru_inblock;         /* block input operations */
        long    ru_oublock;         /* block output operations */
        long    ru_msgsnd;      /* messages sent */
        long    ru_msgrcv;      /* messages received */
        long    ru_nsignals;    /* signals received */
        long    ru_nvcsw;       /* voluntary context switches */
        long    ru_nivcsw;      /* involuntary */
#define ru_last         ru_nivcsw
};
```

The who parameter specifies whose resource usage is to be returned. The resources used by the current process, or by all the terminated children of the current process may be requested.

Resource Limits

The resources of a process for which limits are controlled by the kernel are defined in <sys/resource.h>, and controlled by the getrlimit() and setrlimit() calls:

```
#define RLIMIT_CPU       0    /* cpu time in milliseconds */
#define RLIMIT_FSIZE     1    /* maximum file size */
#define RLIMIT_DATA      2    /* maximum data segment size */
```

```
#define RLIMIT_STACK     3    /* maximum stack segment size */
#define RLIMIT_CORE      4    /* maximum core file size */
#define RLIMIT_RSS       5    /* maximum resident set size */

#define RLIM_NLIMITS     6

#define RLIM_INFINITY    0x7fffffff

struct rlimit {
    int rlim_cur;     /* current (soft) limit */
    int rlim_max;     /* hard limit */
};

int getrlimit(resource, rlp)
int resource;
struct rlimit *rlp;

int setrlimit(resource, rlp)
int resource;
struct rlimit *rlp;
```

Only the super-user can raise the maximum limits. Other users may only alter `rlim_cur` within the range from 0 to `rlim_max` or (irreversibly) lower `rlim_max`.

The `sysconf`(2) interface has been added for POSIX compliance. It allows a process to query the system about system-dependent information.

Memory Locking: `mlock()` and `munlock()`

The `mlock`(3) routine locks selected pages in a process's address space. `munlock()` unlocks selected pages:

```
#include <sys/types.h>

mlock(addr, len)
caddr_t addr; size_t len;

munlock(addr, len)
caddr_t addr; size_t len;
```

3.6. System Operation Support

The call:

```
int swapon(special)
char *special;
```

specifies a device to be made available for paging and swapping. It can be run only by a privileged user.

The call:

```
#include <sys/reboot.h>
reboot(howto, bootargs)
int howto;
char *bootargs;
```

halts or reboots a machine. It too can be run only by a privileged user. The user may request a reboot by specifying `howto` as `RB_AUTOBOOT`, or that the machine be halted with `RB_HALT`. These constants are defined in `<sys/reboot.h>`. `bootargs` is a list of arguments to supply to the `boot(8S)` program.

Accounting

The system optionally keeps an accounting record in a file for each process that exits on the system. The format of this record is beyond the scope of this document. Accounting may be enabled to a file by doing:

```
int acct(path)
char *path;
```

If `path` is null, then accounting is disabled. Otherwise, the named file becomes the accounting file.

3.7. Generic I/O Operations

All filesystem descriptors support the operations `read()`, `write()` and `ioctl()`. We describe the basics of these common primitives here, as well as the `sync()` and `fsync()` primitives. Mechanisms whereby normally synchronous operations may occur in a non-blocking or asynchronous fashion are common to all system-defined abstractions, and are also described here.

read() and write()

The `read()` and `write()` system calls can be applied to communications channels, files, terminals and devices. They have the form:

```
int read(fd, buf, nbytes)
int fd, nbytes;
result caddr_t buf;

int write(fd, buf, nbytes)
int fd, nbytes;
caddr_t buf;
```

The `read()` call transfers as much data as possible from the object defined by `fd` to the buffer at address `buf` of size `nbytes`. `read()` returns the number of bytes transferred, or –1 if the return occurs before any data was transferred because of an error or use of non-blocking operations.

The `write()` call transfers data from the buffer to the object defined by `fd`. Depending on the type of `fd`, it is possible that the `write()` call will accept some portion of the provided bytes; in this case the user should resubmit the other bytes in a later request. Error returns because of interrupted or otherwise incomplete operations are possible.

Scattering of data on input or gathering of data for output is also possible using an array of input/output vector descriptors. The type for the descriptors is defined in `<sys/uio.h>` as:

```
struct iovec {
    caddr_t iov_msg;        /* base of a component */
    int iov_len;            /* length of a component */
};
```

The calls using an array of descriptors are:

```
#include <sys/types.h>
#include <sys/uio.h>

int readv(fd, iov, iovcnt)
int fd;
struct iovec *iov;
int iovcnt;

int writev(fd, iov, iovlen)
int fd,
struct iovec *iov;
int iovlen;
```

Here `iovlen` is the count of elements in the `iov` array. It cannot exceed 16.

Input/Output Control

Control operations on an object are performed by the `ioctl()` operation:

```
ioctl(fd, request, buffer)
int fd, request;
caddr_t buffer;
```

This operation causes the specified `request` to be performed on the object `fd`. The `request` parameter specifies whether the argument buffer is to be read, written, read and written, or is not needed, and also the size of the buffer, as well as the request. Different descriptor types and subtypes within descriptor types may use distinct `ioctl()` requests. For example, operations on terminals control flushing of input and output queues and setting of terminal parameters; operations on disks cause formatting operations to occur; operations on tapes control tape positioning.

The names for basic control operations are defined in `<sys/ioctl.h>`.

Non-Blocking and Multiplexed Operations

A process that wishes to do non-blocking operations on one of its descriptors sets the descriptor in non-blocking mode as described in section *3.4.4*. Thereafter the `read()` call will return a specific `EWOULDBLOCK` error indication if there is no data to be `read()`. The process may `select()` the associated descriptor to determine when a read is possible.

Output attempted when a descriptor can accept less than is requested will either accept some of the provided data, returning a shorter than normal length, or return an error indicating that the operation would block. More output can be performed as soon as a `select()` call indicates the object is writable.

Operations other than data input or output may be performed on a descriptor in a non-blocking fashion. These operations will return with a characteristic error indicating that they are in progress if they cannot complete immediately. The descriptor may then be selected for `write()` to find out when the operation has been completed. When `select()` indicates the descriptor is writable, the operation has completed. Depending on the nature of the descriptor and the operation, additional activity may be started or the new state may be tested.

Asynchronous I/O: `aread()`, `awrite()` and `await()`

Release 4.1 of the SunOS operating system provides the `aread`(3) `awrite`(3) and `await`(3) routines for asynchronous I/O. With these routines, processes that would otherwise block while waiting for a resource can instead proceed with other calculations. Refer to *Writing Device Drivers* for examples of how to use these routines.

File Caches

The call:

```
int fsync(fd)
int fd;
```

moves all modified data and attributes of the file referenced by `fd` to a permanent storage device. When the `fsync()` call returns, all in-memory modified copies of buffers for the associated file have been written to disk. This call is different from `sync()`.

The call:

```
sync()
```

schedules input/output to clean all system buffer caches.

3.8. File System

The file system abstraction provides access to a hierarchical file system structure. The file system contains directories (each of which may contain other subdirectories) as well as files and references to other objects such as devices and inter-process communications sockets.

Each file is organized as a linear array of bytes. No record boundaries or system related information is present in a file. Files may be read and written in a random-access fashion. The user may read the data in a directory as though it were an ordinary file to determine the names of the contained files, but only the system may write into the directories. The file system stores only a small amount of ownership, protection and usage information with a file.

Naming

The file system calls take *pathname* arguments. These consist of a zero or more component filenames separated by / characters, where each filename is up to 255 ASCII characters excluding null and "/".

Each process always has two naming contexts: one for the root directory of the file system and one for the current working directory. These are used by the system in the filename translation process. If a pathname begins with a /, it is called a full pathname and interpreted relative to the root directory context. If the pathname does not begin with a / it is called a relative pathname and interpreted relative to the current directory context.

The system limits the total length of a pathname to 1024 characters.

The filename ".." in each directory refers to the parent directory of that directory. The parent directory of the root of the file system is always that directory.

The calls

```
chdir(path)
char *path;

chroot(path)
char *path;
```

change the current working directory and root directory context of a process. Only the super-user can change the root directory context of a process.

Creation and Removal

The file system allows directories, files and special devices, to be created and removed from the file system.

Directory Creation and Removal

A directory is created with the `mkdir()` system call:

```
int mkdir(path, mode)
char *path;
mode_t mode;
```

where the mode is defined as for files (see below). Note that in Release 4.1, `mkdir()` supports both the Berkeley and the System V group ID semantics. If the set-group-ID bit on a directory is set, objects created within that directory are assigned the GID of that directory, as with the BSD UNIX system. If the GID bit of a parent directory is clear, objects created within it are assigned the GID of the creating process, as in System V.

Directories are removed with the `rmdir()` system call:

```
int rmdir(path)
char *path;
```

A directory must be empty if it is to be deleted.

File Creation

Files are created with the `open()` system call,

```
#include <fcntl.h>

open(path, flag, mode)
int flag, mode;
char *path;
```

The `path` parameter specifies the name of the file to be created. The `flag` parameter must include `O_CREAT` from below to cause the file to be created. The protection for the new file is specified in `mode`. The protection for the new file is specified in `mode`. Bits for `flag` are defined in `<sys/file.h>`:

```
#define O_RDONLY     000      /* open for reading */
#define O_WRONLY     001      /* open for writing */
#define O_RDWR       002      /* open for read & write */
#define O_NDELAY     004      /* non-blocking open */
#define O_APPEND     010      /* append on each write */
#define O_CREAT      01000    /* open with file create */
#define O_TRUNC      02000    /* open with truncation */
#define O_EXCL       04000    /* error on create if file exists */
```

One of `O_RDONLY`, `O_WRONLY` and `O_RDWR` should be specified, indicating what types of operations are desired to be performed on the open file. The operations will be checked against the user's access rights to the file before allowing the `open()` to succeed. Specifying `O_APPEND` causes writes to automatically append to the file. The flag `O_CREAT` causes the file to be created if it does not exist, owned by the current user and the group of the containing directory. The protection for the new file is specified in `mode`. The file mode is used as a three digit octal number. Each digit encodes read access as 4, write access as 2 and execute access as 1, or'ed together. The 700 bits describe owner access, the 070 bits describe the access rights for processes in the same group as the file, and the 007 bits describe the access rights for other processes.

If the open specifies to create the file with `O_EXCL` and the file already exists, then the `open()` will fail without affecting the file in any way. This provides a simple exclusive access facility. If the file exists but is a symbolic link, the open will fail regardless of the existence of the file specified by the link.

Creating References to Devices

The file system allows entries which reference peripheral devices. Peripherals are distinguished as *block* or *character* devices according by their ability to support block-oriented operations. Devices are identified by their *major* and *minor* device numbers. The major device number determines the kind of peripheral it is, while the minor device number indicates one of possibly many peripherals of that kind. Structured devices have all operations performed internally in "block" quantities while unstructured devices often have a number of special `ioctl()` operations, and may have input and output performed in varying units. The `mknod()` call creates special entries:

```
int mknod(path, mode, dev)
char *path;
int mode, dev;
```

where `mode` is formed from the object type and access permissions. The parameter `dev` is a configuration dependent parameter used to identify specific character or block I/O devices.

A new interface to mknod(), mkfifo() has been provided for POSIX compliance. mkfifo() creates a named pipe.

File and Device Removal

A reference to a file or special device may be removed with the unlink() call,

```
int unlink(path)
char *path;
```

The caller must have write access to the directory in which the file is located for this call to be successful.

Reading and Modifying File Attributes

Detailed information about the attributes of a file system may be obtained with the calls:

```
#include <sys/vfs.h>

int statfs(path, buf)
char *path;
struct statfs *buf;

int fstatfs(fd, buf)
int fd;
struct statfs *buf;
```

The statfs structure includes the file system type, file system block size, total blocks in the file system, free blocks, free blocks available to non-super-user, total file nodes in the file system, free file nodes in the file system, and the file system ID.

Directory entries can be obtained in a filesystem-independent format by using the getdents() call:

```
#include <sys/types.h>
#include <sys/dirent.h>

int getdents(fd, buf, nbytes)
int fd;
char *buf;
int nbytes;
```

Detailed information about the attributes of a file may be obtained with the calls:

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(path, stb)
char *path;
struct stat *stb;

fstat(fd, stb)
int fd;
struct stat *stb;
```

The `stat` structure includes the file type, protection, ownership, access times, size, and a count of hard links. If the file is a symbolic link, then the status of the link itself (rather than the file the link references) may be found using the `lstat()` call:

```
int lstat(path, stb)
char *path;
result struct stat *stb;
```

Newly created files are assigned the UID of the process that created them and the GID of the directory in which they are created. The ownership of a file may be changed by either of the calls

```
#include <sys/types.h>

int chown(path, owner, group)
char *path;
uid_t owner;
gid_t group;

int fchown(fd, owner, group)
int fd;
uid_t owner;
gid_t group;
```

In addition to ownership, each file has three levels of access protection associated with it. These levels are owner relative, group relative, and global (all users and groups). Each level of access has separate indicators for read permission, write permission, and execute permission. The protection bits associated with a file may be set by either of the calls:

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(path, mode)
char *path;
mode_t mode;

int fchmod(fd, mode)
int fd, mode;
```

where `mode` is a value indicating the new protection of the file as listed above in the *File Creation* section.

Three additional bits exist: the 04000 "set-user-ID" bit can be set on an executable file to cause the EUID of a process which executes the file to be set to the owner of that file; the 02000 bit has a similar effect on the EGID. The 01000 bit causes an image of an executable program to be saved longer than would otherwise be normal; this "sticky" bit is a hint to the system that a program is heavily used.

Finally, the access and modify times on a file may be set by the call:

```
#include <sys/types.h>

int utimes(file, tvp)
char *file;
struct timeval *tvp;
```

This is particularly useful when moving files between media, to preserve relationships between the times the file was modified.

Links and Renaming

Links allow multiple names for a file to exist.

Two types of links exist, *hard* links and *symbolic* (sometimes called "soft") links. A hard link is a reference counting mechanism that allows a file to have multiple names within the same file system. Symbolic links cause string substitution during the pathname interpretation process. Unlike hard links, symbolic links can exist independently of the file being linked to.

Hard links and symbolic links have different properties. A hard link insures the target file will always be accessible, even after its original directory entry is removed; no such guarantee exists for a symbolic link. Symbolic links can span file systems boundaries.

The following calls create a new link, named `path2`, to `path1`:

```
int link(path1, path2)
char *path1, *path2;

int symlink(path1, path2)
char *path1, *path2;
```

The `unlink()` primitive may be used to remove either type of link.

If a file is a symbolic link, the ''value'' of the link may be read with the `readlink()` call,

```
int readlink(path, buf, bufsiz)
char *path, *buf;
int bufsiz;
```

This call returns, in `buf`, the null-terminated string substituted into pathnames passing through `path`.

Atomic renaming of file system resident objects is possible with the `rename()` call:

```
int rename(oldname, newname)
char *oldname, *newname;
```

where both `oldname` and `newname` must be in the same file system. If `newname` exists and is a directory, then it must be empty.

Two new interfaces for file system queries have been provided for POSIX compliance. pathconf(2) and fpathconf() answer questions about the named file and/or the underlying file system. These routines always return properly with 4.1 and later UFS file systems. NFS® file systems that are served by a server recognizing mount protocol version 2 can also provide this information for the NFS files. The NFS file system must be mounted with the posix option.

Extension and Truncation

Files are created with zero length and may be extended simply by writing or appending to them. While a file is open the system maintains a pointer into the file indicating the current location in the file associated with the descriptor. This pointer may be moved about in the file in a random access fashion. To set the current offset into a file, the lseek() call may be used,

```
#include <sys/types.h>
#include <sys/unistd.h>

off_t lseek(fd, offset, whence)
int fd;
off_t offset;
int whence;
```

where whence is given in <sys/file.h> as one of,

```
#define L_SET    0    /* set absolute file offset */
#define L_INCR   1    /* set file offset relative to current position */
#define L_XTND   2    /* set offset relative to end-of-file */
```

The call:

```
lseek(fd, 0, L_INCR)
```

returns the current offset into the file.

Files may have ''holes'' in them. Holes are void areas in the linear extent of the file where data has never been written. These may be created by seeking to a location in a file past the current end-of-file and writing. Holes are treated by the system as zero-valued bytes.

A file may be truncated (or extended) with either of the calls:

```
int truncate(path, length)
char *path;
off_t length;

int ftruncate(fd, length)
int fd;
off_t length;
```

The truncate() and ftruncate() system calls set the length of a file. If the newly specified length is shorter than the file's current length, the file is shortened. However, if the new length is longer, the file's size is increased to the desired length. When writing a file exclusively through mapped access, `truncate()` and `ftruncate()` are the only alternatives to `MAP_RENAME` operations for growing a file.

Checking Accessibility

A process running with different real and effective user ids may interrogate the accessibility of a file to the real user by using the `access()` call:

```
int access(path, mode)
char *path;
int mode;
```

Here `mode` is constructed by taking the logical OR of the following bits, defined in `<sys/file.h>`:

```
#define F_OK     0    /* file exists */
#define X_OK     1    /* file is executable */
#define W_OK     2    /* file is writable */
#define R_OK     4    /* file is readable */
```

The presence or absence of advisory locks does not affect the result of `access()`.

File Locking

The file system provides basic facilities that allow cooperating processes to synchronize their access to shared files. A process may place an advisory `read()` or `write()` lock on a file, so that other cooperating processes may avoid interfering with the process' access. This simple mechanism provides locking with file granularity. The system does not force processes to obey locks placed by `flock()`; they are of an advisory nature only. Locks placed by `flock()` are only visible to processes running on the local processor.

Locking is performed after an `open()` call by applying the `flock()` primitive:

```
flock(fd, operation)
int fd, operation;
```

where the `operation` parameter is formed from bits defined in `<sys/file.h>`:

```
#define LOCK_SH 1    /* shared lock */
#define LOCK_EX 2    /* exclusive lock */
#define LOCK_NB 4    /* don't block when locking */
#define LOCK_UN 8    /* unlock */
```

Successive lock calls may be used to increase or decrease the level of locking. If an object is currently locked by another process when a `flock()` call is made, the caller will be blocked until the current lock owner releases the lock; this may be avoided by including `LOCK_NB` in the `operation` parameter. Specifying

LOCK_UN removes all locks associated with the descriptor. Advisory locks held by a process are automatically deleted when the process terminates.

File and Record Locking: lockf()

The lockf(3) routine allows you to lock a specified record (set of contiguous bytes), or an entire file. The file must be write-accessible by the process. Locks placed by lockf() are visible to any process running on any processor with access to the file:

```
#include <unistd.h>

int lockf(fd, cmd, size)
int fd, cmd;
long size;
```

The cmd argument can be one of:

```
#define  F_ULOCK  0  /* Unlock a previously locked section */
#define  F_LOCK   1  /* Lock a section for exclusive use */
#define  F_TLOCK  2  /* Test and lock a section (non-blocking)
#define  F_TEST   3  /* Test section for other process' locks
```

The size argument indicates the number of bytes in the segment to lock; the segment starts at the current offset within the file. If size is zero, lockf() places a lock on the segment from the current offset through the end of the file (so a call to lockf() immediately after an open() would lock the entire file).

Mounting Filesystems

The call:

```
int mount(type, dir, flags, data)
char *type, *dir;
int flags;
caddr_t data;
```

extends the UNIX name space. The mount() call specifies a block device type containing a UNIX file system to be made available starting at dir. If flags is set then the file system is read-only; writes to the file system will not be permitted and access times will not be updated when files are referenced. data is a pointer to a structure which contains the type specific arguments to mount.

The call:

```
unmount(dir)
char *dir;
```

unmounts the file system mounted on dir. umount() call will succeed only if the file system is not currently being used.

Disk Quotas

As an optional facility, each file system may be requested to impose limits on a user's disk usage. Two quantities are limited: the total amount of disk space which a user may allocate in a file system and the total number of files a user may create in a file system. Quotas are expressed as *hard* limits and *soft* limits. A hard limit is always imposed; if a user would exceed a hard limit, the operation which caused the resource request will fail. A soft limit results in the user receiving a warning message, but with allocation succeeding. Facilities are provided to turn soft limits into hard limits if a user has exceeded a soft limit for an unreasonable period of time.

To manipulate disk quotas on a file system the `quotactl()` call is used:

```
#include <ufs/quota.h>

int quotactl(cmd, special, uid, addr)
int cmd, uid;
char *special;
caddr_t addr;
```

where `cmd` indicates a command to be applied to the UID. `special` is a pointer to a null-terminated string containing the path name of the block special device for the file system being manipulated. The block special device must be mounted. `addr` is the address of an optional, command specific, data structure which is copied in or out of the system. The interpretation of `addr` is given with each command.

3.9. Devices

The system uses a collection of device drivers to access attached peripherals. Such devices are generally grouped into two classes: structured devices on which block-oriented input/output operations occur (basically disks and tapes), and unstructured devices (anything else).

Structured Devices

Structured devices include disk and tape drives, and are accessed through a system buffer-caching mechanism, which permits them to be accessed as ordinary files, by means of random-access reads and writes.

The `mount`(8) command in the system allows a structured device containing a file system volume to be accessed through the operating system.

Tape drives also typically provide a structured interface, although this is rarely used.

Unstructured Devices

Unstructured devices are those devices which do not support a randomly accessed block structure.

Communications lines, raster plotters, normal magnetic tape access (in large or variable size blocks), and access to disk drives permitting large block transfers and special operations like disk formatting and labeling all use unstructured device interfaces.

Much more information about device drivers can be found in *Writing Device Drivers.*

3.10. Debugging Support

ptrace() provides a means by which a process may control the execution of another process, and examine and change its memory image. Its primary use is for the implementation of breakpoint debugging.

```
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

ptrace(request, pid, addr, data, addr2)
enum ptracereq request;
int pid, data;
char *addr, *addr2;
```

There are five arguments whose interpretation depends on the `request` argument. Generally, `pid` is the PID of the traced process. A process being traced behaves normally until it encounters some signal whether internally generated like "illegal instruction" or externally generated like "interrupt." See `sigvec(2)` for the list. Then the traced process enters a stopped state and the tracing process is notified via `wait(2)`. When the traced process is in the stopped state, its memory image can be examined and modified using `ptrace()`. If desired, another `ptrace()` request can then cause the traced process either to terminate or to continue, possibly ignoring the signal.

Note that several different values of the `request` argument can make `ptrace()` return data values — since –1 is a possibly legitimate value, to differentiate between –1 as a legitimate value and –1 as an error code, you should clear the `errno` global error code before doing a `ptrace()` call, and then check the value of `errno` afterwards.

The value of the `request` argument determines the precise action of the call:

`PTRACE_TRACEME`
: This request is the only one used by the traced process; it declares that the process is to be traced by its parent. All the other arguments are ignored. Peculiar results will ensue if the parent does not expect to trace the child.

`PTRACE_PEEKTEXT, PTRACE_PEEKDATA`
: The word in the traced process's address space at `addr` is returned. `addr` must be even (except on Sun386i machines), the child must be stopped and the input `data` and `addr2` are ignored.

`PTRACE_PEEKUSER`
: The word of the system's per-process data area corresponding to `addr` is returned. `addr` must be a valid offset within the kernel's per-process data structures. This space contains the registers and other information about the process; its layout corresponds to the `user` structure in the system.

`PTRACE_POKETEXT, PTRACE_POKEDATA`
: The given `data` is written at the word in the process's address space corresponding to `addr`, which must be even (except on Sun386i machines). No useful value is returned. If the instruction and data spaces are separate request `PTRACE_PEEKTEXT` indicates instruction space while

PTRACE_PEEKDATA indicates data space. The PTRACE_POKETEXT request must be used to write into a process's text space even if the instruction and data spaces are not separate.

PTRACE_POKEUSER
The process's system data is written, as it is read with request PTRACE_PEEKUSER. Only a few locations can be written in this way: the general registers, the floating point status and registers, and certain bits of the processor status word.

PTRACE_CONT
The data argument is taken as a signal number and the child's execution continues at location addr as if it had incurred that signal. Normally the signal number will be either 0 to indicate that the signal that caused the stop should be ignored, or that value fetched out of the process's image indicating which signal caused the stop. If addr is (int *)1 then execution continues from where it stopped.

PTRACE_KILL
The traced process terminates.

PTRACE_SINGLESTEP
Execution continues as in request PTRACE_CONT; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. On Sun machines the T-bit is used and just one instruction is executed.

PTRACE_ATTACH
Attach to the process identified by the pid argument and begin tracing it. Process pid does not have to be a child of the requester, but the requester must have permission to send process pid a signal and the effective userids of the requesting process and process pid must match.

PTRACE_DETACH
Detach the process being traced. Process pid is no longer being traced and continues its execution. The data argument is taken as a signal number and the process continues at location addr as if it had incurred that signal.

PTRACE_GETREGS
The traced process's registers are returned in a structure pointed to by the addr argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in <machine/reg.h> describes the data that is returned.

PTRACE_SETREGS
The traced process's registers are written from a structure pointed to by the addr argument. The registers include the general purpose registers, the program counter and the program status word. The 'regs' structure defined in <machine/reg.h> describes the data that is set.

PTRACE_READTEXT, PTRACE_READDATA
Read data from the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates

instruction space while PTRACE_READDATA indicates data space. The addr argument is the address within the traced process from where the data is read, the data argument is the number of bytes to read, and the addr2 argument is the address within the requesting process where the data is written.

PTRACE_WRITETEXT, PTRACE_WRITEDATA
Write data into the address space of the traced process. If the instruction and data spaces are separate, request PTRACE_READTEXT indicates instruction space while PTRACE_READDATA indicates data space. The addr argument is the address within the traced process where the data is written, the data argument is the number of bytes to write, and the addr2 argument is the address within the requesting process from where the data is read.

As indicated, these calls (except for requests PTRACE_TRACEME and PTRACE_ATTACH) can be used only when the subject process has stopped. The wait() call is used to determine when a process stops; in such a case the 'termination' status returned by wait has the value WSTOPPED to indicate a stop rather than genuine termination.

To forestall possible fraud, ptrace() inhibits the set-user-ID and set-group-ID facilities on subsequent execve(2) calls. If a traced process calls execve(), it will stop before executing the first instruction of the new image showing signal SIGTRAP.

4

Networking Overview

This chapter provides an overview of the socket-based and Transport Layer Interface-based Interprocess Communication (IPC) facilities, along with the Internet and RPC-based network services in Release 4.1 of the SunOS operating system.

4.1. Socket-Based Interprocess Communications

Interprocess Communication Primitives

This chapter introduces the socket-based interprocess communications facilities that the SunOS operating system has adapted from BSD. Much more detail about these facilities can be found in part three of *Network Programming*. For an introduction to the networking facilities which Sun has added to its system in the time since socket-based IPC was developed, see the *Network Services* section of this same *Network Programming* manual. (These facilities include the *Network File System*, the *Remote Procedure Call* mechanisms, and the *External Data Representation* standard). For detailed information about AT&T-style STREAMS, see *STREAMS Programming*.

Communication Domains

The system provides access to an extensible set of communication *domains*. A communication domain is identified by a manifest constant defined in the file `<sys/socket.h>`. Important standard domains supported by the system are the UNIX domain, `AF_UNIX`, for communication within the system, and the "internet" domain for communication with the DARPA Internet protocol family, `AF_INET`. Other domains can be added to the system.

Socket Types and Protocols

Within a domain, communication takes place between endpoints known as *sockets*. Each socket has the potential to exchange information with other sockets of an appropriate type within the domain.

Each socket has an associated abstract type, which describes the semantics of communication using that socket. Properties such as reliability, ordering, and prevention of duplication of messages are determined by the type. The basic set of socket types is defined in `<sys/socket.h>`:

```
/* Standard socket types */
#define SOCK_DGRAM      1    /* datagram */
#define SOCK_STREAM     2    /* virtual circuit */
#define SOCK_RAW        3    /* raw socket */
#define SOCK_RDM        4    /* reliably-delivered message */
#define SOCK_SEQPACKET  5    /* sequenced packets */
```

The SOCK_DGRAM type models the semantics of datagrams in network communication: messages may be lost or duplicated and may arrive out-of-order. A datagram socket may send messages to and receive messages from multiple peers. The SOCK_RDM type models the semantics of reliable datagrams: messages arrive unduplicated and in-order, the sender is notified if messages are lost. The send() and receive() operations (described below) generate reliable/unreliable datagrams. The SOCK_STREAM type models connection-based virtual circuits: two-way byte streams with no record boundaries. Connection setup is required before data communication may begin. The SOCK_SEQPACKET type models a connection-based, full-duplex, reliable, sequenced packet exchange; the sender is notified if messages are lost, and messages are never duplicated or presented out-of-order. Users of the last two abstractions may use the facilities for out-of-band transmission to send out-of-band data.

SOCK_RAW is used for unprocessed access to internal network layers and interfaces; it has no specific semantics.

Other socket types can be defined.

Each socket may have a concrete *protocol* associated with it. This protocol is used within the domain to provide the semantics required by the socket type. Not all socket types are supported by each domain; support depends on the existence and the implementation of a suitable protocol within the domain. For example, within the "internet" domain, the SOCK_DGRAM type may be implemented by the UDP user datagram protocol, and the SOCK_STREAM type may be implemented by the TCP transmission control protocol, while no standard protocols to provide SOCK_RDM or SOCK_SEQPACKET sockets exist.

Socket Creation, Naming, and Service Establishment

Sockets may be *connected* or *unconnected.* An unconnected socket descriptor is obtained by the socket() call:

```
s = socket(domain, type, protocol);
    result int s;
    int domain, type, protocol;
```

The socket domain and type are as described above, and are specified using the definitions from <sys/socket.h>. The protocol may be given as 0, meaning any suitable protocol. One of several possible protocols may be selected using identifiers obtained from a library routine, getprotobyname().

An unconnected socket descriptor of a connection-oriented type may yield a connected socket descriptor in one of two ways: either by actively connecting to another socket, or by becoming associated with a name in the communications domain and *accepting* a connection from another socket. Datagram sockets need not establish connections before use.

To accept connections or to receive datagrams, a socket must first have a binding to a name (or address) within the communications domain. Such a binding may be established by a `bind()` call:

```
bind(s, name, namelen);
    int s, namelen;
    struct sockaddr *name;
```

Datagram sockets may have default bindings established when first sending data if not explicitly bound earlier. In either case, a socket's bound name may be retrieved with a `getsockname()` call:

```
getsockname(s, name, namelen);
    int s;
    result struct sockaddr *name;
    result int *namelen;
```

while the peer's name can be retrieved with `getpeername()`:

```
getpeername(s, name, namelen);
    int s;
    result struct sockaddr *name;
    result int *namelen;
```

Domains may support sockets with several names.

Accepting Connections

Once a binding is made to a connection-oriented socket, it is possible to `listen()` for connections:

```
listen(s, backlog);
    int s, backlog;
```

The *backlog* specifies the maximum count of connections that can be simultaneously queued awaiting acceptance.

An `accept()` call:

```
t = accept(s, name, anamelen);
    result int t, *anamelen;
    int s;
    result struct sockaddr *name;
```

returns a descriptor for a new, connected, socket from the queue of pending connections on *s*. If no new connections are queued for acceptance, the call will wait for a connection unless non-blocking I/O has been enabled.

Making Connections

An active connection to a named socket is made by the `connect()` call:

```
connect(s, name, namelen);
    int s, namelen;
    struct sockaddr *name;
```

Although datagram sockets do not establish connections, the `connect()` call may be used with such sockets to create an *association* with the foreign address. The address is recorded for use in future `send()` calls, which then need not supply destination addresses. Datagrams will be received only from that peer, and asynchronous error reports may be received.

It is also possible to create connected pairs of sockets without using the domain's name space to rendezvous; this is done with the `socketpair()` call[2]:

```
socketpair(domain, type, protocol, sv);
    int domain, type, protocol;
    result int sv[2];
```

Here the returned *sv* descriptors correspond to those obtained with `accept()` and `connect()`.

The call

```
pipe(pv);
    result int pv[2];
```

creates a pair of `SOCK_STREAM` sockets in the UNIX domain, with `pv[0]` only writable and `pv[1]` only readable.

Sending and Receiving Data

Messages may be sent to a socket by:

```
cc = sendto(s, buf, len, flags, to, tolen);
    result int cc;
    int s, len, flags, tolen;
    caddr_t buf, to;
```

if the socket is not connected or:

```
cc = send(s, buf, len, flags);
    result int cc;
    int s, len, flags;
    caddr_t buf;
```

if the socket is connected. The corresponding receive primitives are:

[2] This release supports `socketpair()` creation only in the "unix" communication domain.

```
msglen = recvfrom(s, buf, len, flags, from, fromlenaddr);
    result int *fromlenaddr;
    result int msglen;
    int s, len, flags;
    result caddr_t buf, from;
```

and

```
msglen = recv(s, buf, len, flags);
    result int msglen;
    int s, len, flags;
    result caddr_t buf;
```

In the unconnected case, the parameters *to* and *tolen* specify the destination or source of the message, while the *from* parameter stores the source of the message, and **fromlenaddr* initially gives the size of the *from* buffer and is updated to reflect the true length of the *from* address.

All calls cause the message to be received in or sent from the message buffer of length *len* bytes, starting at address *buf*. The *flags* specify peeking at a message without reading it or sending or receiving high-priority out-of-band messages, as follows:

```
#define MSG_PEEK    0x1 /* peek at incoming message */
#define MSG_OOB 0x2 /* process out-of-band data */
```

Scatter/Gather and Exchanging Access Rights

It is possible to scatter and gather data and to exchange access rights with messages. When either of these operations is involved, the number of parameters to the call becomes large. Thus the system defines a message header structure, in `<sys/socket.h>`, which is used to contain the parameters to the calls:

```
struct msghdr {
    caddr_t msg_name;        /* optional address */
    int msg_namelen;         /* size of address */
    struct  iov *msg_iov;    /* scatter/gather array */
    int msg_iovlen;          /* # elements in msg_iov */
    caddr_t msg_accrights;   /* access rights sent/received */
    int msg_accrightslen;    /* size of msg_accrights */
};
```

Here *msg_name* and *msg_namelen* specify the source or destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter/gather locations, as described in section *3.7.1*. Access rights to be sent along with the message are specified in *msg_accrights*, which has length *msg_accrightslen*. In the “unix” domain these are an array of integer descriptors, taken from the sending process and duplicated in the receiver.

This structure is used in the operations `sendmsg()` and `recvmsg()`:

```
sendmsg(s, msg, flags);
    int s, flags;
    struct msghdr *msg;

msglen = recvmsg(s, msg, flags);
    result int msglen;
    int s, flags;
    result struct msghdr *msg;
```

Using **read()** and **write()** with Sockets

The normal `read()` and `write()` calls may be applied to connected sockets and translated into `send()` and `receive()` calls from or to a single area of memory and discarding any rights received. A process may operate on a virtual circuit socket, a terminal or a file with blocking or non-blocking input/output operations without distinguishing the descriptor type.

Shutting Down Halves of Full-Duplex Connections

A process that has a full-duplex socket such as a virtual circuit and no longer wishes to read from or write to this socket can give the call:

```
shutdown(s, direction);
    int s, direction;
```

where *direction* is 0 to not read further, 1 to not write further, or 2 to completely shut the connection down. If the underlying protocol supports unidirectional or bidirectional shutdown, this indication will be passed to the peer. For example, a shutdown for writing might produce an end-of-file condition at the remote end.

Socket and Protocol Options

Sockets, and their underlying communication protocols, may support *options*. These options may be used to manipulate implementation specific or protocol-specific facilities. The `getsockopt()` and `setsockopt()` calls are used to control options:

```
getsockopt(s, level, optname, optval, optlen);
    int s, level, optname;
    result caddr_t optval;
    result int *optlen;

setsockopt(s, level, optname, optval, optlen);
    int s, level, optname; caddr_t optval; int optlen;
```

The option *optname* is interpreted at the indicated protocol *level* for socket *s*. If a value is specified with *optval* and *optlen*, it is interpreted by the software operating at the specified *level*. The *level* `SOL_SOCKET` is reserved to indicate options maintained by the socket facilities. Other *level* values indicate a particular protocol which is to act on the option request; these values are normally interpreted as a "protocol number".

UNIX Domain

This section describes briefly the properties of the UNIX communications domain.

Types of Sockets

In the UNIX domain, the `SOCK_STREAM` abstraction provides pipe-like facilities, while `SOCK_DGRAM` provides datagrams — unreliable message-style communications.

Naming

Socket names are strings and the current implementation of the UNIX domain embeds bound sockets in the file system name space; this is a side effect of the implementation.

Access Rights Transmission

The ability to pass descriptors with messages in this domain allows migration of service within the system and allows user processes to be used in building system facilities.

Internet Domain

This section describes briefly how the Internet domain is mapped to the model described in this section. More information will be found in the *Networking Implementation Notes* section of *Network Programming.*

Socket Types and Protocols

`SOCK_STREAM` is supported by the Internet TCP protocol; `SOCK_DGRAM` by the UDP protocol. Each is layered atop the transport-level Internet Protocol (IP). The Internet Control Message Protocol is implemented atop/beside IP and is accessible via a raw socket.

Socket Naming

Sockets in the Internet domain have names composed of the 32 bit internet address, and a 16 bit port number. Options may be used to provide IP source routing or security options. The 32-bit address is composed of network and host parts; the network part is variable in size and is frequency encoded. The host part may optionally be interpreted as a subnet field plus the host on subnet; this is enabled by setting a network address mask at boot time.

Access Rights Transmission

No access rights transmission facilities are provided in the Internet domain.

Raw Access

The Internet domain allows the super-user access to the raw facilities of IP. These interfaces are modeled as `SOCK_RAW` sockets. Each raw socket is associated with one IP protocol number, and receives all traffic received for that protocol. This allows administrative and debugging functions to occur, and enables user-level implementations of special-purpose protocols such as inter-gateway routing protocols.

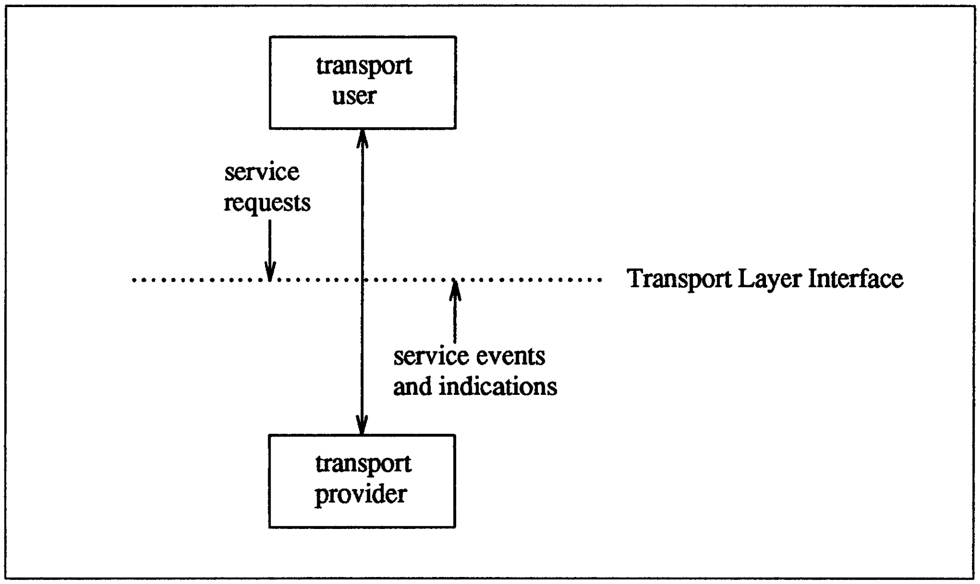
4.2. TLI Communication Facilities

This section gives an overview of the Transport Layer Interface, which supports the transfer of data between two processes in a manner compatible with System V Release 3.

TLI uses an architecture similar to that of sockets as described above. Communication takes place between a transport *provider*, and a transport *user*.

An example of a transport provider is the TLI-based TCP transport protocol. A transport user may be a networking application or session-layer protocol.

Figure 4-1 *Transport Layer Interface*



The transport user accesses the service by issuing the appropriate requests. One example is a request to transfer data over a connection. Similarly, the provider notifies the user of various events, such as the arrival of data on a connection.

Modes of Service

TLI provides two modes of service, connection-mode and connectionless-mode.

Connection-mode is circuit-oriented and enables data to be transmitted over an established connection in a reliable, sequenced manner (akin to TCP over sockets). Connection-mode also provides an identification mechanism that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-mode, in contrast, is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units (akin to UDP). This service requires only a preexisting association between the peer users involved, which determines the characteristics of the data to be transmitted. All the information required to deliver a unit of data (for example, the destination address) is presented to the transport provider, together with the data to be transmitted, in one service access (which need not relate to any other service access). Each unit of data transmitted is entirely self-contained.

Connectionless-mode service is attractive for applications that:

- involve short-term request/response interactions
- exhibit a high level of redundancy
- are dynamically reconfigurable
- do not require guaranteed, in-sequence delivery of data

Connection-Mode Service

Connection-mode transport service is characterized by four phases:

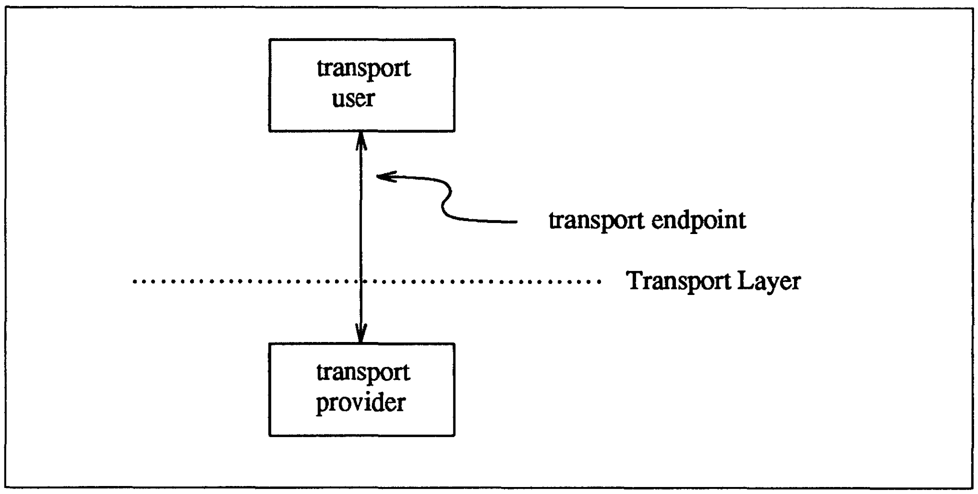
- local management
- connection establishment
- data transfer, and
- connection release.

Local Management

The local management phase defines local operations between a transport user and a transport provider. For example, a user must establish a channel of communication with the transport provider, as illustrated below. Each channel between a transport user and transport provider is a unique endpoint of communication, and will be called the transport endpoint.

The `t_open`(3N) routine enables a user to choose a particular transport provider that will supply the connection-mode services, and establishes the transport endpoint.

Figure 4-2 *Channel Between User and Provider*



Another necessary local function for each user is to establish an identity with the transport provider. Each user is identified by a transport address. More accurately, a transport address is associated with each transport endpoint, and one user process may manage several transport endpoints. In connection-mode service, one user requests a connection to another user by specifying that user's address. The structure of a transport address is defined by the address space of the transport provider. An address may be as simple as a random character string (for example, ''file_server''), or as complex as an encoded bit pattern that specifies all information needed to route data through a network. Each transport provider defines its own mechanism for identifying users. Addresses may be assigned to each transport endpoint by `t_bind`(3N)

Table 4-1 *Local Management Routines*

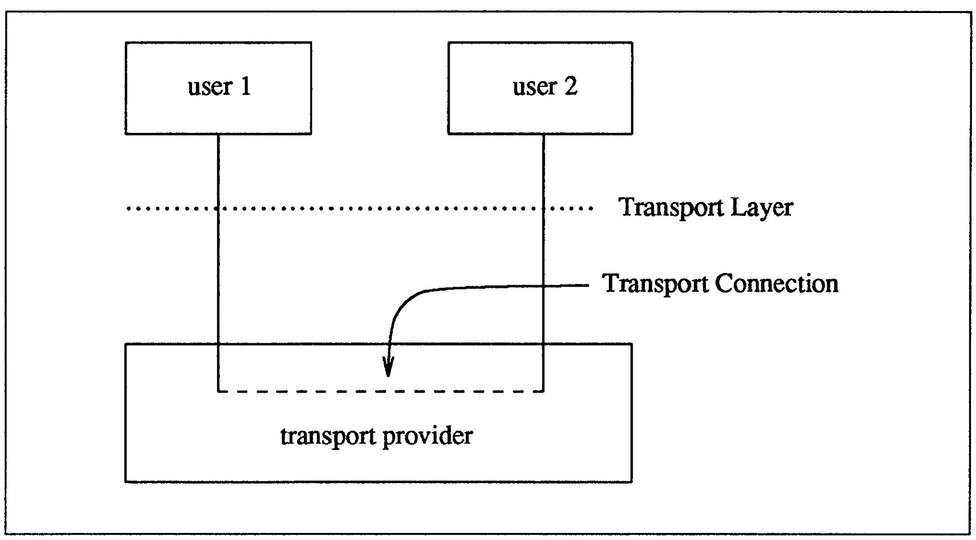
Routine	Description
`t_alloc()`	Allocates TLI data structures.
`t_bind()`	Binds a transport address to a transport endpoint.
`t_close()`	Closes a transport endpoint.
`t_error()`	Prints an error message.
`t_free()`	Frees structures allocated using `t_alloc()`.
`t_getinfo()`	Returns a set of parameters associated with a particular transport provider.
`t_getstate()`	Returns the state of a transport endpoint.
`t_look()`	Returns the current event on a transport endpoint.
`t_open()`	Establishes a transport endpoint connected to a chosen transport provider.
`t_optmgmt()`	Negotiates protocol-specific options with the transport provider.
`t_sync()`	Synchronizes a transport endpoint with the transport provider.
`t_unbind()`	Unbinds a transport address from a transport endpoint.

In addition to `t_open()` and `t_bind()`, several routines are available to support local operations. The table below summarizes the TLI local management routines.

Connection Establishment

The connection establishment phase enables two users to create a connection, or virtual circuit, between them, as shown below.

Figure 4-3 *Transport Connection*



This phase is illustrated by a client-server relationship between two transport users. One user, the server, typically advertises some service to a group of users, and then listens for requests from those users. As each client requires the service, it attempts to connect itself to the server using the server's advertised transport address. The t_connect(3N) routine initiates the connect request. One argument to t_connect(), the transport address, identifies the server the client wishes to access. The server is notified of each incoming request using t_listen(3N) and may call t_accept(3N) to accept the client's request for access to the service. If the request is accepted, the transport connection is established.

The next table summarizes all routines available for establishing a transport connection.

Table 4-2 *Connection Establishment Routines*

Routine	Description
t_accept()	Accepts a request for a transport connection.
t_connect()	Establishes a connection with the transport user at a specified destination.
t_listen()	Retrieves an indication of a connect request from another transport user.
t_rcvconnect()	Completes connection establishment if t_connect() was called in asynchronous mode.

Data Transfer

The data transfer phase enables users to transfer data in both directions over an established connection. Two routines, t_snd(3N) and t_rcv(3N) send and receive data over this connection. All data sent by a user is guaranteed to be delivered to the user on the other end of the connection in the order in which it was sent. The table below summarizes the connection mode data transfer routines.

Table 4-3 *Connection Mode Data Transfer Routines*

Routine	Description
t_rcv()	Retrieves data that has arrived over a transport connection.
t_snd()	Send data over an established transport connection.

Connection Release

The connection release phase provides a mechanism for breaking an established connection. When you decide that the conversation should terminate, you can request that the provider release the transport connection. TLI supports two types of connection release. The first is an abortive release, which directs the transport provider to release the connection immediately. Any previously sent data that has not yet reached the other transport user may be discarded by the transport provider. The t_snddis(3N) routine initiates this abortive disconnect, and t_rcvdis(3N) processes the incoming indication of an abortive disconnect.

All transport providers must support the abortive release procedure. In addition, some transport providers may also support an orderly release facility that enables users to terminate communication gracefully with no data loss. The functions t_sndrel(3N) and _rcvrel(3N) support this capability, as shown below.

Table 4-4 *Connection Release Routines*

Routine	Description
t_rcvdis()	Returns an indication of an aborted connection, including a reason code and user data.
t_rcvrel()	Returns an indication that the remote user has requested an orderly release of a connection.
t_snddis()	Aborts a connection or rejects a connect request.
t_sndrel()	Requests the orderly release of a connection.

Connectionless-Mode Service

The connectionless-mode transport service is characterized by two phases: local management and data transfer. The local management phase defines the same local operations described above for the connection-mode service.

The data transfer phase enables a user to transfer data units (sometimes called datagrams) to the specified peer user. Each data unit must be accompanied by the transport address of the destination user. Two routines, t_sndudata(3N) and t_rcvudata(3N). support this message-based data transfer facility. The table below summarizes all routines associated with connectionless-mode data transfer.

Routine	Description
t_rcvudata()	Retrieves a message sent by another transport user.
t_rcvuderr()	Retrieves error information associated with a previously sent message.
t_sndudata()	Sends a message to the specified destination user.

State Transitions

In addition to library routines that provide transport services to users TLI also provides state transition rules that define the sequence in which the transport routines may be invoked. These transition rules take the form of state tables, which are explained in detail in *Network Programming*. TLI state tables define the legal sequence of library calls based on state information and the handling of events. These events include user-generated library calls, as well as provider-generated event indications.

For more information about TLI-based communication, refer to *Network Programming*.

4.3. Network-Based Services

Release 4.1 is considerably more sophisticated than the first versions of the UNIX system. This is true not only in terms of programming environments and tools, though 4.1 does include many networking features from 4.3 BSD and virtually all System V Release 3 networking facilities. Release 4.1 is oriented, at a fundamental level, to networks of closely linked machines. It is *structurally* a network system, and is designed to evolve with the evolution of computer network technology.

Derived from networking features in 4.2 BSD, network services were implemented with special-purpose daemons (server processes) working in close cooperation with the kernel, rather than in the kernel itself. Release 4.1 continues this line of development. Its network services, from the Network File System (NFS) and Remote Execution Facility (REX) to its network name service serviceypname are built upon a server-based architecture.

When a network service is added to the system, it is added by means of a server process which is executed on all machines providing the service. Each server then communicates with the kernel or with its peers on other machines as necessary. Sun servers do differ in one very significant way from those which were inherited from BSD, they are usually based on Sun's Remote Procedure Call (RPC) mechanism. As a consequence, they automatically benefit from the features provided by RPC and the External Data Representation (XDR), protocol, including the data portability provided by XDR and the modularity of RPC's authentication system.

There are a number of benefits to a server-based approach to the provision of network services:

- The kernel itself remains more manageable in size and complexity, and more clearly delimited in function. Its job is to implement the virtual machine on the system that hosts it. It does not negotiate with other machines for the non-local resources that it needs.
- When network services are implemented as independent server processes, they are easily tuned and controlled.
- They can be invoked only when needed (see `inetd`(8)) and thus consume no run-time resources when not in use. And they are easily updated to accommodate protocol and transport changes. Indeed, when such changes are made, multiple versions of the same server can be run simultaneously, thus allowing development to proceed without rendering old applications obsolete.

The overall effect is thus an *extensible* environment in which new network services can be easily added to the system by building upon XDR, RPC, network communications and other services. Network services, then, are analogous to commands: anyone can add one to effectively extend the system.

NOTE *See the Network Services section of Network Programming* for more information about the fundamental network services.

4.4. Standard Server-Based Services

Networking functions contained within the kernel include the network and transport levels of the system networking support, the network device drivers, the IP and TCP protocol code and the NFS itself. Other network services are provided by server processes:

`/usr/etc/biod`
Block I/O daemon. Used by an NFS client to handle read-ahead and write-behind for blocks in the buffer cache.

/usr/etc/bootparams
NFS boot daemon. Provides the information that diskless clients need for booting. If the yp name service isn't available, it consults the bootparams database, /etc/bootparams.

/usr/etc/in.comsat
Listens to a non-standard UDP socket used for incoming mail notification, as enabled by the biff program.

/usr/etc/rpc.etherd
etherd collects, summarizes and reports statistics on packet traffic for a given network interface.

/usr/etc/in.fingerd
in.fingerd provides support for the ARPA-standard finger command, which displays information about the current users of a given machine.

/usr/etc/in.ftpd
File Transfer Protocol daemon. This is the ARPA standard file transfer protocol.

/usr/etc/inetd
Opens sockets for all the servers listed in /etc/inetd.conf, and then starts them up when requests are made on them.

/usr/etc/keyserv
The DES authentication daemon. Stores secret keys and controls access to them. keyserv will not talk to anything but a local root process.

/usr/etc/rpc.lockd
The network lock manager daemon. Provides System V compatible advisory file and record locking for both local and NFS mounted files.

/usr/etc/rpc.mountd
NSF mount daemon. Handles mount requests for files systems exported over the NFS.

/usr/etc/in.named
named is the Internet domain name server.

/usr/etc/nfsd
Network File System daemon. The real work is done in the kernel by way of a magic system call that never returns.

/usr/etc/portmap
Demultiplexes UDPs for Remote Procedure Calls, converting RPC program numbers to DARPA protocol port numbers.

/usr/etc/rarpd
rarpd is a daemon that responds to Reverse-ARP requests.

/usr/etc/rpc.rexd
rexd is the RPC server that controls remote program execution.

/usr/etc/in.rexecd
rexecd is the server for the rexec() routine. It provides remote

execution facilities with authentication based on user names and passwords.

/usr/etc/in.rlogind
Remote Login daemon.

/usr/etc/rmt
Remote magnetic tape access. Used by the remote dump and restore programs to manipulate a tape driver over the network.

/usr/etc/in.routed
Routing table update daemon. Uses a non-standard UDP protocol to update kernel routing tables.

/usr/etc/rpc.rquotad
`rquotad` returns quotas for a user of a local file system which is mounted by a remote machine over the NFS. The results are used by `quota` to display remote file systems user quotas.

/usr/etc/in.rshd
Remote shell daemon. Non-standard TCP protocol to allow remote execution with authentication based on privileged port numbers.

/usr/etc/rpc.rusersd
Remote user daemon. Necessary to support the `rusers` command.

/usr/etc/rpc.rwalld
Remote write-to-all daemon. Handles `rwall` and `shutdown` requests.

/usr/etc/in.rwhod
Remote `who` daemon. Generates broadcasts periodically about the status of logged-in users, and listens to the broadcasts of other servers on the local network and maintains the database that is printed by `rwho`. Not used much in the Sun environment since the protocol involves lots of broadcast packets.

/usr/lib/sendmail
Provides mail transport through the Simple Mail Transfer Protocol (SMTP).

/usr/etc/rpc.sprayd
Spray daemon. Used by the `spray` command for network diagnosis.

/usr/etc/rpc.rstatd
Remote status daemon. The primary purposes for this server are returning kernel performance statistics for `perfmeter`, and responding to requests from `rup`.

/usr/etc/in.syslog
Reads a datagram (UDP) socket and logs information it receives according to a configuration file.

/usr/etc/in.talkd
Listens on a UDP port, and negotiates talk TCP connections. This protocol doesn't even work between Vaxes and Suns.

/usr/etc/in.telnetd
The ARPA-standard remote terminal service.

/usr/etc/tfsd
Translucent file-system daemon. Provides copy-on-write access to a private overlay of a read-only file system. Refer to ADMIN for details.

/usr/etc/in.tftpd
Trivial file transfer protocol daemon. Can be used for simple, non-authenticated file transfers. Also used to load boot files.

/usr/etc/in.timed
The ARPA-standard time service. Note that this service only provides the system time to clients who request it, and is not a full network synchronization service.

/usr/etc/in.tnamed
The `tnamed` daemon supports the old obsolete DARPA Name Server Protocol.

/usr/etc/ypbind
`ypbind` remembers information that lets client processes on a single host communicate with some `ypserv` process. It must run on every machine which has yp name service client processes.

/usr/etc/rpc.yppasswdd
Runs on yp name service masters only. Supports password change requests for the yp name service password database.

/usr/etc/ypserv
Runs on all yp name service servers. The `ypserv` daemon's primary function is to look up information in the local yp name service database.

/usr/etc/rpc.ipallocd
(Sun386i only). The `rpc.ipallocd` daemon maps Ethernet addresses to IP addresses, allocating temporary IP addresses when necessary.

/usr/etc/rpc.pnpd
(Sun386i only). The `rpc.pnpd` daemon configures new systems onto a Sun386i network, and distributes configuration information for systems already on the network. It also provides configuration RPC calls for diskless clients.

5

Programmer's Guide to Security Features

This chapter is for system programmers interested in writing secure programs for the Release 4.1 of the SunOS operating system. The first section below discusses system calls from a security standpoint, and the second section discusses C library routines from this standpoint. The remaining sections give practical advice on writing secure C programs.

5.1. System Calls

System calls provide entry points into the operating system kernel. When a program makes a system call, the kernel itself services the request. When a program calls a library routine, it's just like calling a function defined in the program, except the function is defined in a system library. Library routines may or may not employ system calls. System calls are documented in Section 2 of the *SunOS Reference Manual*; library routines are documented in Section 3 of that manual.

I/O Routines

There are four basic I/O operations: creating a file, opening a file, reading, and writing. Descriptions follow:

`creat()`

This call creates a new file, or recreates an old file zero-length. It takes two arguments indicating the file's name and its mode:

```
creat("/tmp/data", 0644);
```

`creat` returns a valid file descriptor, or –1 if there was an error. The process must have write and execute permission for the directory where the file is being created. The file's owner and group are set to the effective user ID and group ID. The file's permissions are set according to the second argument, modified by the default file creation mask `umask`.

`open()`

This call opens a file for reading and writing, or both. It takes two or three arguments indicating the file's name, the input/output combination, and the mode (as above). `open()` returns a valid file descriptor, or –1 if the process doesn't have proper access permissions. Once a process opens a file, changing permissions on that file and its containing directories does not affect the original access permissions.

`read()`

This call reads data from a file previously opened by `open()`, which deals with all access permissions.

`write()` This call writes data to a file previously opened by `open()`, which deals with all access permissions.

Process Control

There are three basic process control operations: forking a new process, overlaying this process with an executable image, and signaling a process.

`fork()` This call creates a new process (the child) that is an exact copy of the calling process (the parent). All processes on the system are created this way. Here are some security considerations:

- The child inherits the real and effective user and group IDs.
- The child inherits the default file mode creation mask, `umask`.
- All open files are passed to the child.

`exec*()` These calls copy an executable program into the space occupied by the calling process.† Generally this is done after forking a new process, so as not to destroy the parent. All programs on the system are executed this way. Here are some security considerations:

- The real and effective user and group IDs are normally inherited by an executed program.
- However, the effective user ID (or group ID) is set to the owner (group) of the executed program, if the program has the set user ID (set group ID) bit turned on.
- The new program inherits the default file mode creation mask, `umask`.
- All open files (except those with the close-on-exec flag) are passed to the new program.

`signal()` This call provides an exception and interrupt handling facility. It takes two arguments: the number (or name) of a signal, and the action to take when that signal occurs. If the action is `SIG_IGN`, the signal is ignored; if it is `SIG_DFL`, the signal is handled in the default manner; if it is the name of a function, that function gets executed on receipt of signal. The `lockscreen` program ignores most signals, for example, so that it can't be stopped or killed by an unfriendly user. Many programs trap interrupts so they can delete temporary files.

File Attributes

Three system calls affect the permissions and ownership/group of a file. Two more system calls return the accessibility and attribute status of a file.

`umask()` This call sets the default file creation mask for the calling process and all its children. It takes one argument, just as with the `umask` command.

`chmod()` This call changes the permission modes of a file or directory. It takes two arguments: the file name and the numeric mode, as with the `chmod` command.

`chown()` This call changes both the owner and the group of a specified file. It takes three arguments: the file name, the numeric user ID, and the group number. In this

† Actually only `execve()` is a system call; the others – `execl()`, `execv()`, `execle()`, `execlp()`, `execvp()` – are library routines.

sense it is a combination of the chown and chgrp commands. Note that the chown() system call turns off both setuid and setgid permission, for security reasons. This is so these permissions do not get given out by mistake.

access() This call determines the accessibility of a file. It takes two arguments: the name of the file in question, and the type of access to be tested (specified as an integer between 0 and 7).

0	the file exists
1	it is executable
2	it is writable
3	writable and executable
4	it is readable
5	readable and executable
6	readable and writable
7	readable, writable, and executable

These numbers are exactly the same as the modes for chmod(1). Note that access() uses real (instead of effective) user ID and group ID to determine accessibility. This property makes it useful inside setuid and setgid programs, which alter only the effective user and group IDs.

stat() This call returns the attribute status of a file. It takes two arguments: the name of the file in question, and the address of a stat structure, defined in <sys/stat.h>. This status structure contains the following information, among other things:

st_dev	ID of the device containing the file
st_ino	i-node number of the file
st_mode	type and permission mode
st_nlink	number of links
st_uid	user ID of the file's owner
st_gid	group ID of the file's group
st_size	size of the file in bytes
st_atime	last access time (read)
st_mtime	last modification time (write)
st_ctime	last status change (to i-node)

Note that the -l option of the ls command prints the modification time, not the atime or ctime.

User ID and Group ID

A set of system calls permits C programs to get and set both real and effective user and group IDs.

getuid() This call returns the real user ID of a process. Programs may employ this call inside setuid programs to determine which user has really invoked a program.

getgid() This call returns the real group ID of a process. Programs may employ this call inside setgid programs to determine the original group of the invoker.

geteuid() This call returns the effective user ID of a process. Programs that should have the setuid permission bit turned on can employ this call to verify that they are in fact running setuid. Also, programs can employ this call to determine if they are running `setuid` to some other user than the one who invoked it.

`getegid()` This call returns the effective group ID of a process. Programs that should have the `setgid` permission bit turned on can employ this call to verify that they are in fact running `setgid`. Also, programs can employ this call to determine if they are running `setgid` to some other group than that of the invoker.

`setreuid()` This call sets either the real or the effective user ID, or both. It takes two arguments: the real user ID, and the effective user ID. When either argument is –1, that value is not changed. If the effective user ID of the calling process is:

- Super-user, both real and effective user IDs can be set to any legal value.
- Not super-user, the real user ID can be set to the effective user ID, or the effective user ID can be set to the real user ID or to the saved set-user ID from `execve`(2).

Programs can toggle between real and effective user IDs by exchanging them, using this system call or the `seteuid()` library routine.

`setgroups()` This call, which is restricted to the super-user, sets the group access list of the current process. It takes two arguments: the number of groups, and a pointer to an array of integers specifying numeric group IDs.

5.2. C Library Routines

Library routines are system services that offer programs the advantage of convenience and reliability. Many library routines make use of system calls, discussed above. The C library is documented in section 3 of the reference manual, while system calls are documented in section 2.

Standard I/O

The Standard I/O Library is the most commonly used set of routines for reading and writing files.

`fopen()` This call opens a file for reading or writing, or both. It creates a file if necessary. Security considerations are the same as those for `open()`.

Reading The `fread()`, `fgetc()`, `getc()`, `fgets()`, `gets()`, `fscanf()`, and `scanf()` routines read information from a file opened by `fopen()`, or from standard input. Once a file stream is open for reading, it remains readable even if its access permissions change.

Writing The `fwrite()`, `fputc()`, `putc()`, `fputs()`, `fprintf()`, and `printf()` routines write information to a file opened by `fopen()`, or to standard output. Once a file stream is open for writing, it remains writable even if its access permissions change.

`system()` This call runs `/usr/bin/sh` to execute the command specified as its argument. Try to avoid making this call inside a `setuid root` program, as the invoked shell has super-user permission.

`popen()` This call invokes the command specified as its argument using `fork()` and `exec()`, then creates a pipe to the new process using `pipe()`. Be extremely careful when making this call inside a `setuid root` program, as the spawned process has super-user permission.

Password Processing

Several library routines are available for reading system password files and for dealing with passwords typed at the terminal.

`getpass()`

This call prints its argument (a prompt) on the terminal, turns off echoing, then reads a password typed at the terminal, up to eight characters long. It returns a pointer to the password string. This routine is often used in conjunction with `crypt()` to obtain an encrypted password.

`getpwnam()`

Given a login name, this call returns a pointer to a `passwd` structure, filled with the corresponding password file entry. This structure is defined in `<pwd.h>` and looks like this:

```
struct  passwd {
    char    *pw_name;
    char    *pw_passwd;
    int     pw_uid;
    int     pw_gid;
    int     pw_quota;
    char    *pw_comment;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
};
```

On C2 secure systems, the `pw_passwd` field does not contain an encrypted password, but rather an indication that the encrypted password resides somewhere else.

`getpwuid()`

Given a numeric user ID, this call returns a pointer to a `passwd` structure, filled with the corresponding password file entry.

`getpwent()`

This call is used for sequential processing of the password file. Initially it opens the file and returns the first entry. Thereafter it returns the following entry. The related `setpwent()` call rewinds the password file, and the `endpwent()` call closes the password file.

`putpwent()`

This call is used to change or extend the `/etc/passwd` file. Here are the steps involved in this process:

1. Create a unique temporary file such as `/etc/pw$$` where the `$$` represents
2. Link the temporary file to the conventional temporary file `/etc/ptmp`. If the link fails, remove the unique temporary file and exit; somebody else is modifying the password file.
3. Read from `/etc/passwd` with successive calls to `getpwent()`, and write to `/etc/ptmp` with successive calls to `putpwent()`, making changes as necessary.
4. Move `/etc/passwd` to a backup file such as `/etc/opasswd`.
5. Link `/etc/ptmp` to `/etc/passwd`.
6. Unlink the two temporary files, `/etc/ptmp` and `/etc/pw$$`.

At this point no library routines are available for dealing gracefully with the /etc/security/passwd.adjunct file on C2 secure systems. Fortunately there should be little reason to tamper with this file anyway. Because password entries for most users are stored in the YP Name Service, the putpwent() routine is of limited utility, in any case.

Group Processing

A set of routines is available to deal with the /etc/group file, analogous to the routines just described.

getgrnam()

Given a group name, this call returns a pointer to a group structure, filled with the corresponding group file entry. This structure is defined in <grp.h>.

getgrgid()

Given a numeric group ID, this call returns a pointer to a group structure, filled with the corresponding group file entry.

getgrent()

This call is used for sequential processing of the group file. Initially it opens the file and returns the first entry. Thereafter it returns the following entry. The related setgrent() call rewinds the group file, and the endgrent() call closes the group file. In a defeat of symmetry, there exists no putgrent() library routine.

Who's Running a Program?

The most reliable method of determining who is running a program is to employ getuid() along with getpwuid(). The first call returns the real user ID, which gets handed to the second call so it can look up the user's login name.

```
#include <pwd.h>
    .
    .
    .
    struct passwd *pwent;

    pwent = getpwuid(getuid());
    printf("User name is %s\n", pwent->pw_name);
```

There are other methods of determining a user's identity, but they aren't as reliable as the code above.

getlogin()

This call is supposed to return a pointer to the name of the user logged into a terminal. The routine examines standard input, output, and error (in order), in case they are redirected. The first associated with a terminal produces a terminal name, which is used to find an associated user name in /etc/utmp. If a process was run by at, it has no associated terminal, so getlogin() returns a null pointer. Unfortunately getlogin() can be fooled by changing the terminal associated with standard input, for example with this Bourne shell command:

```
$ program 0> /dev/tty07
```

This would cause a getlogin() call inside program to return the name of the user logged into /dev/tty07. As a consequence, the use of getlogin() is discouraged.

Encryption Routines

NOTE *These encryption routines are only available in the U.S.A. by way of the Domestic Encryption Kit.*

In 1977, the National Bureau of Standards announced an encryption method "for use in [unclassified applications on] Federal ADP systems and networks," called DES (Data Encryption Standard). This encryption method uses a 56-bit key to perturb 8 bytes of data at a time. Because the key was shortened from 128 bits (as recommended by IBM) to 56 bits, DES can be attacked by brute force – trying all possible keys – but the computation required takes a long time even on a supercomputer. As a consequence, DES is relatively secure, because it costs so much to break.

Release 4.1 libraries offer a set of routines implementing DES, using hardware if it is available, which can be used to encrypt and decrypt sensitive data. In addition, there is an older set of routines used mainly for encrypting passwords, employing a modified DES that has not been implemented in hardware. These routines are used for password encryption to prevent hardware assistance for breaking into the system.

The `des_crypt` Library

This DES encryption library is faster and more general purpose than the older encryption routines based on `encrypt()`. Furthermore, the `des_crypt` library employs DES hardware when it is available. Programs using the newer library must include `<des_crypt.h>`. Two flavors of encryption are available: Electronic Code Book (ECB) mode, which encrypts blocks of data independently, and Cipher Block Chaining (CBC) mode, which chains together successive blocks. The second mode is more secure, because it protects against insertions, deletions, and substitutions, and also because regularities in clear text do not appear in cipher text.

`des_setparity()` This routine should be called first to set the parity of the 8-byte encryption key. This call takes a single argument: a character pointer, whose contents get modified. Note that in DES, the parity bit is the low bit (not the high bit) of each byte.

`ecb_crypt()` This routine implements Electronic Code Book mode. It takes four arguments: the encryption key discussed above, a character pointer to the data involved, an unsigned integer indicating the data's length, and an unsigned integer indicating the mode of operation. Flags are ORed into the mode as necessary: `DES_ENCRYPT` means to encrypt, `DES_DECRYPT` means to decrypt, and `DES_HW` means to use DES hardware if available. The `ecb_crypt()` routine returns an integer status code.

`cbc_crypt()` This routine implements Cipher Block Chaining mode. It takes five arguments: the encryption key discussed above, a character pointer to the data involved, an unsigned integer indicating the data's length, an unsigned integer indicating the mode of operation, and a character pointer to an 8-byte initialization vector for chaining. At first the initialization vector should be zeroed out, but afterwards it gets updated to the next initialization vector on each call. Flags are ORed into the mode as necessary: `DES_ENCRYPT` means to encrypt, `DES_DECRYPT`

means to decrypt, and DES_HW means to use DES hardware if available. The cbc_crypt() routine returns an integer status code.
Note that these library routines are used by the des command, discussed in the previous chapter.

Password Encryption Routines

The older and slower DES encryption routines based on encrypt() are used primarily for encrypting passwords. The password encryption routine crypt() involves a "salt" used to perturb the encrypting algorithm, so that DES chips cannot be used to assist in cracking login passwords. Furthermore, this routine calls encrypt() sixteen times to eat up CPU cycles. If a cryptanalyst wanted to search the key space for miniscules – trying all possible 8-letter combinations of lowercase letters – it would take about 3000 years on a Sun-3. Allowing for combinations of uppercase letters and digits as well, it would take much longer. That's why guessing a password is a more efficient way to break security than searching the key space.

setkey() Given a 64-byte character array of ones and zeros (8 bytes worth of text), this routine creates the 56-bit DES encryption key, which is used by the following routine to encrypt or decrypt text.

encrypt() This routine encrypts or decrypts a 64-byte character array of ones and zeros specified as the first argument (8 bytes worth of text), according to whether the second argument is zero (meaning encrypt) or one (meaning decrypt).

crypt() This call is used to encrypt an 8-letter password, usually obtained from getpass(), presented above. This call takes two arguments: a character pointer to the typed password (the key), and a character pointer to a two-letter salt for perturbing the algorithm. The salt string may be longer, but only the first two characters are relevant. First crypt() hands the key to setkey(), and then calls encrypt() repeatedly. Finally crypt() returns a pointer to the encrypted password. Here's how crypt() is typically used in a C program:

```
#include <pwd.h>
    .
    .
    .
    char *username, *p, *passwd, *getpass(), *crypt();
    struct passwd *pwd;

    if ((pwd = getpwnam(username)) == NULL) {
        fprintf(stderr, "No such user name.\n");
        exit(1);
    }
    p = getpass("password:");
    passwd = crypt(p, pwd->pw_passwd);
    if (strcmp(passwd, pwd->pw_passwd)) {
        fprintf(stderr, "Incorrect password.\n");
        exit(2);
    }
```

Note: the crypt() library routine should not be confused with the crypt shell command, which uses a much less sophisticated encoding algorithm, one that can be broken by brute force in several hours of CPU time. Users seeking a higher level of security can always use the more secure `des` shell command, however.

User and Group ID

These library routines allow programs to set user and group ID, both real and effective. The first routine behaves differently if compiled with the System V compatibility library rather than with the standard C library.

`setuid()`

This call sets both the real and effective user ID of the current process to the specified numeric user ID. The super-user may set real and effective user IDs to any value; other users may set them only if the argument is the real or effective user ID.

When programs are compiled using the System V compatibility library, this call sets the real user ID and/or the effective user ID to the specified numeric user ID. The super-user may set both the real and effective user IDs to any value. Other users may set only the effective user ID, and only if the specified argument is the same as the real user ID, or if the argument is the same as the saved set-user ID from `exec()`. This arrangement permits toggling between real and effective user IDs.

`seteuid()`

This call sets the effective user ID of the current process to the specified numeric user ID. The super-user may set the effective user ID to any value; other users may set it only if the argument is the real user ID.

`setruid()`

This call sets the real user ID of the current process to the specified numeric user ID. The super-user may set the real user ID to any value; other users may set it only if the argument is the effective user ID.

`setgid()`

This call sets both the real and effective group ID of the current process to the specified numeric group ID. The super-user may set real and effective group IDs to any value; other users may set them only if the argument is the real or effective group ID.

`setegid()`

This call sets the effective group ID of the current process to the specified numeric group ID. The super-user may set the effective group ID to any value; other users may set it only if the argument is the real group ID.

`setrgid()`

This call sets the real group ID of the current process to the specified numeric group ID. The super-user may set the real group ID to any value; other users may set it only if the argument is the effective group ID.

5.3. Writing Secure Programs

When you're trying to write secure C programs, there are two important guidelines you should follow:

1. Make sure that temporary files created by the program don't contain sensitive information that isn't encrypted. When in doubt, store data in memory. Also, verify that temporary files are readable and writable only by the owner. It's always a good idea to call `umask(077)` at the beginning of a program. Also, it's best to create temporary files in private directories that are writable only by the owner. However, if you must use `/tmp`, get your system administrator to set its mode to 2777 (set group ID) so that files in it may be deleted only by their owner.

2. Make sure that any command the program runs – whether with `exec()`, `system()`, or `popen()` – is the command that should be run, and not a Trojan horse. This is especially important if your program is `setuid` or `setgid`, in which case programs should always reset the user ID before running any commands.

Let's look at some ways a program can be fooled into running a Trojan horse. In this innocent-looking function call, the `vi` command invoked is the first one in the search path. If a user copied `/usr/bin/csh` to `$HOME/bin/vi`, and had `$HOME/bin` as the first element of `PATH`, the program would actually invoke that user's private copy of the C shell, not the `vi` command:

```
system("vi");
```

This is because `system()` inherits the `PATH` environment from the program, which inherits it from the user's login shell. The logical way to avoid this potential problem, it seems, would be to specify the full path name:

```
system("/usr/bin/vi");
```

This can be circumvented as well. All a clever user has to do is move the purloined C shell `$HOME/bin/vi` to `$HOME/bin/bin`, write a shell script named `vi` in the current directory, and modify the shell and environment variable `IFS` (input field separator) to slash. In this case, `system()` thinks the command above means to run `$HOME/bin/bin` with the argument `vi`. The logical way to avoid this further problem is to set `IFS` before invoking the command:

```
system("IFS=' \t\n'; export IFS; /bin/vi");
```

That looks pretty cluttered, but is nearly impossible to crack. A further problem arises if the command is to be invoked with argument. Clever users could put command separators such as ampersand or semicolon into the argument list, followed by invocations of `/usr/bin/csh` or something similar. In `setuid root` programs, that C shell would also run `setuid root`, giving the cracker full access to the system. The only solution to this potential problem is to parse arguments before passing them to a program.

Set User ID Programs

Any programs you write that are `setuid` **must** reset the user ID before invoking any commands. Here's the easiest way to do this:

```
int saveid;

saveid = geteuid();
setuid(getuid());
system("/usr/bin/ed");
setuid(saveid);
```

For this to work properly, you must use the System V compatibility library by compiling with /usr/5bin/cc instead of /usr/bin/cc. Without the System V compatibility library, it is impossible to set the effective user ID back to what it was when a setuid program was first invoked.

Set Group ID Programs

The same cautions apply to programs that set group ID, as to programs that set user ID. Any programs you write that are setgid **must** reset the group ID before invoking any commands. Here's the easiest way to do this:

```
int saveid;

saveid = getegid();
setgid(getgid());
system("/usr/bin/ed");
setgid(saveid);
```

To work properly this also requires the System V compatibility library, so use /usr/5bin/cc to compile.

Commands with Shell Escapes

Be wary of commands that allow shell escapes, such as mail, write, dc, edit, ex, vi, ed, sed, awk, troff, and perhaps others. Make especially sure that programs never call these commands while in setuid or setgid mode. See the examples above.

Shell Scripts and Security

The same caveats apply to shell scripts as to C programs. Whenever a shell script involves sensitive data or affects system security, you should be careful to set the input field separators and the search path before proceeding with the guts of the script:

```
IFS=" ^I
"
PATH=/bin:/usr/bin
export IFS PATH
```

setuid or setgid, shell scripts are potential security risks for the user or group, and should be avoided if possible (or restricted in scope to a particular file system using).chroot(8) When such scripts are used, it is even more important to set IFS and PATH before proceeding.

Shell scripts that are setuid to root should *never* be used.

Guidelines for Secure Programs

Here are some guidelines for writing secure setuid and setgid programs.

1. Don't do it unless absolutely necessary.
2. Set the group ID rather than the user ID. It's best to create a new special-purpose group, but if that's impossible, don't use a system group. When you use an existing group, remember that you may be compromising files that belong to other users in the group.
3. Don't exec() any commands. Remember that the library calls system() and popen() call some form of exec().

4. If you must `exec()` a command, set the effective group ID to the real group ID first with `setgid(getgid())`.
5. If you can't reset the effective group ID, set the `IFS` when calling `system()` or `popen()`, and invoke a command using its full pathname.
6. Don't pass user-specified arguments to `system()` or `popen()`. If you must, check user-specified arguments for special shell characters.
7. If you have a large program that must execute a lot of other programs, don't make it `setgid` – write a smaller, simpler `setgid` program and execute it from the large program.
8. If you must set user ID instead of group ID, remember that all of the above also applies to `setuid` permission.
9. Don't make a program set user ID to `root`. Pick another login, or better yet create another login, but don't use `root`.

Here are some guidelines for installing `setuid` and `setgid` programs.

1. Make sure a `setuid` or `setgid` command is not writable by group or others. Never set the mode to anything less restrictive than 4755 (for `setuid` commands) or 2755 (for `setgid` commands).
2. Better yet, set the modes to 4111 (for `setuid` commands) or 2111 (for `setgid` commands) so that snoopers can't run the `strings` command on the binary to search for security holes.
3. Be wary of programs that come from unknown sources. Search through the code for calls to `exec()`, ,`system()` and `popen()`. If a program is supposed to be installed `setuid` or `setgid`, read the source code closely. Never install such a program unless you get source code.
4. Pay close attention when installing new software. Some `make/install` procedures create `setuid` and `setgid` programs indiscriminately. Programs should never employ `root` privileges merely to change the owner or group of a file, since this can be done without being super-user. Check for commands that may create `setuid` files, such as these:

```
cp su /tmp/su
cp /usr/bin/csh /tmp/su
```

5.4. Programming as Superuser

This section describes considerations for programs to be run only by `root`, and for programs that absolutely must be made `setuid root`.

Some system calls are restricted to processes whose effective user ID is `root`. Also, many routines presented earlier in this chapter behave differently when called by the super-user than when called by an ordinary user. Furthermore, the system does not perform permission checks if the user is `root`. The super-user is always allowed access. For example, `open()` does not check the permissions of a file when called by `root` – it simply opens the file. This lack of checking makes being super-user very dangerous.

Commands run by the super-user are `root` processes (except for a non-`root` `setuid` program, which has the effective user ID of the program's owner). Furthermore, `setuid` `root` programs, and commands executed from within one, are also `root` processes.

`setuid()` When called from a `root` process, this call sets both the effective and the real user ID, rather than just the effective user ID. This is allowed so that users can log in to the system. After the system boots up, the `init` process spawns a `getty` process for each terminal; when `getty` reads a login name, it calls `login` to read and validate the password. Since all three processes run as `root`, `login` is able to set the real and effective user IDs for a user's shell. Once a process loses `root` permission, it can't get it back. Thus programs should get privileged operations out of the way before calling `setuid()`.

`setgid()` When called from a `root` process, this call sets both the effective and the real group ID. Unlike `setuid()`, which only sets the user ID to a valid number, `setgid()` set the group ID to any integer, whether or not that value is associated with a group.

`chown()` When run by a `root` process, this routine does not remove `setuid` or `setgid` permission. When run by a non-`root` process, however, such permissions are removed.

`chroot()` This system call changes a process' idea of where the root directory is. After this call, a process cannot change directory above the new root, and all path searches begin at the new root directory. This call is useful for setting up restricted environments. Obviously, only `root` processes are allowed to perform this operation.

`mknod()` This system call is used to create special files, such as device drivers. Aside from FIFOs (named pipes), only `root` can run this call successfully. Most programs never use this call because special files can be created with the administrative command `/etc/mknod`.

Security considerations for the system calls `mount()` and `umount()` are described in the chapter on system administration.

6

Native Language Application Support

6.1. Introduction

Sun's native language application support features allow developers to create applications that are readily portable between various native languages. Users and developers both benefit when applications can be installed without change between locales having different languages and customs.

Portability between native languages can substantially reduce a user's difficulties when configuring applications for different locales. It also allows for international distribution of standard applications, while simplifying the problems of training and support. While the language representation may change, the program's internal operations do not. This portability is also referred to as *internationalization.*

Portability across languages greatly simplifies the process of adapting versions of an application to fit local markets. This adaptation process is also referred to as *localization.*

Overview

Release 4.1 of the SunOS operating system provides support for developing and executing applications that operate in native languages whose characters are included in the ISO 8859/1 (ISO Latin 1) character set. These include most major European languages, such as: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Spanish and Swedish.

Readers interested in Asian language environments should refer to the *Japanese Language Environment Product Description*, Part Number 800-3148-10.

The native language application support features in Release 4.1 are an integral part of the operating system's command and programmatic interface. They encompass:

- A common data model based on the ISO Latin 1 code set (with added support for multi-byte characters).
- Commands that operate cleanly on that model (8-bit clean commands).
- I/O device support for ISO Latin 1 characters, including native-language keyboards, a *compose* key to produce composite characters not found on a given keyboard, on-screen fonts, and (optionally) printer support.
- A standard announcement mechanism that allows users to select or change language environments (*locales*) when using native-language applications. When provided, users may select a native language environment base for a

given host, or they may choose different locales for different applications on the same system. The base locale supplied with 4.1 is the "C" environment, as described in Volume 3 of the *X/OPEN Programmer's Guide, Issue 2*; (XPG2); 4.1 provides facilities for developing and installing other locales.

- Programming support, including 8-bit clean library routines, routines that make use of language-specific character collation orders, conversion schemes, and format conventions, and routines that produce language-specific (diagnostic) messages.

Standards-Based Approach

The traditional approach of many computer vendors has been to adopt proprietary solutions for international applications. However, those solutions would only operate on a particular vendor's installed base. By contrast, the standard-based internationalization features in Release 4.1 support portability across differing native language environments as well as different vendor platforms.

The approach used in Release 4.1 is compatible with the internationalization routines described in the ANSI X3.159-1989 C language standard. It is based on the NLS system described in XPG2. Since 4.1 conforms to XPG2, and also includes the ANSI C internationalization routines, XPG3-compliant applications can readily be ported to 4.1.

4.1 also conforms to the IEEE Standard 1003.1 (POSIX.1). For more information about X/OPEN compatibility and POSIX conformance, refer to the chapters, *X/OPEN Compatibility Features* and *POSIX Conformance*, respectively, in this manual.

Common Data Model

Prior to Release 4.1, the SunOS operating system did not support a common method for representing characters in the various European languages. Applications that required the use of characters other than those in the 7-bit US ASCII character set (see `ascii`(7)) were forced to provide proprietary (non-standard) methods to represent and operate on them. Thus, text produced by one internationalized application might well be unusable by another, and would almost certainly be unusable with system commands and library routines based on the 94 characters allowed with 7-bit ASCII.

The ISO Latin 1 character set uses 8 bits to represent each character, allowing for 188 characters. It is compatible with 7-bit ASCII in that the encodings for the printable ASCII characters are the same (8th bit set to 0). For purposes of text representation, ISO Latin 1 can be thought of as a superset of ASCII. For a listing of this character set, refer to Appendix A, *ISO Latin 1 Character Set*.

The ability to represent the characters of many languages using this common character set allows applications operating in different native languages to communicate with each other.

8-Bit Clean Commands

To support the notion of a native-language application environment, a number of commands used to process user input (text) have been modified to support 8-bit characters. Prior to 4.1, many system commands were "8-bit dirty," meaning that they interpreted ASCII control characters (those with the eighth bit set to 1) in specialized ways. Some simply masked off the eighth bit, while others used it as a flag of some sort.

Other than those listed in the table below, all commands in 4.1 can be regarded as 8-bit clean. That is, they either support 8-bit character data, or are not concerned with processing text.

Table 6-1 *8-Bit Dirty Commands*

8-Bit Dirty Commands					
adb	cpp	keylogin	man	rusers	users
addbib	ctags	keylogout	newgrp	rwho	w
as	cxref	lex	nroff	sdb	who
awk	dbx	lint	passwd	spell	whoami
catman	dbxtool	login	refer	strings	whois
cc†	deroff	logname	rlogin	su	yacc
cflow	dis	m4	rmail	troff	

†Supports 8-bit characters in strings and comments.

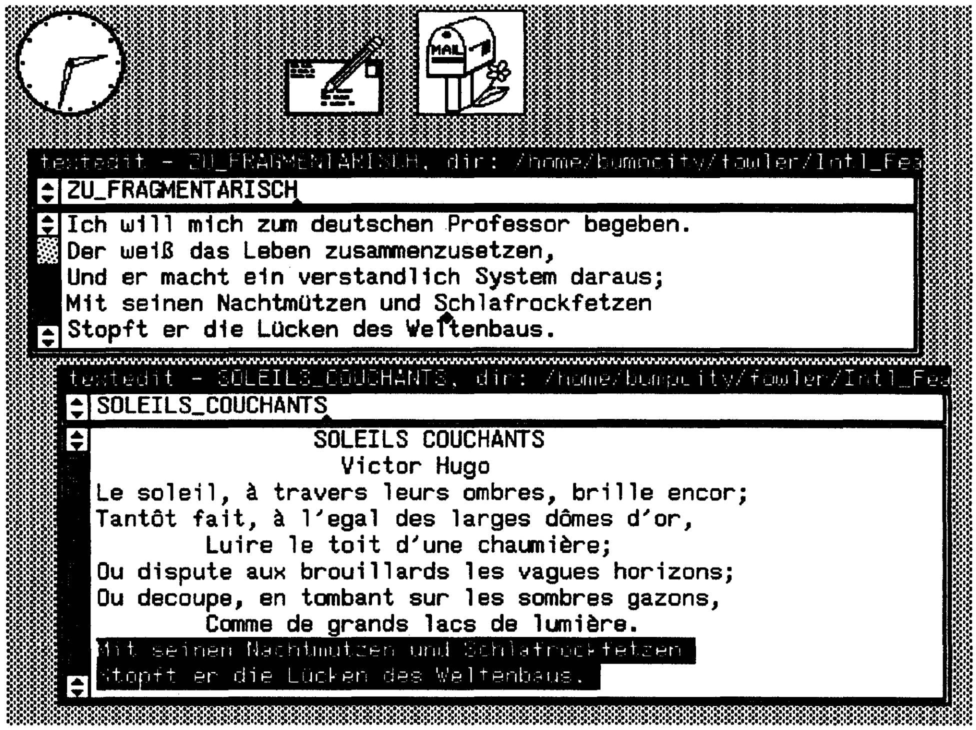
I/O Device Support

SunView 1

The screen fonts provided with SunOS 4.1 can display the entire range of ISO Latin 1 characters.

SunView 1.8, bundled with Release 4.1 handles the input, editing and screen display of native language characters. All the SunView based desktop tools, such as mailtool, textedit, commandtool and others, provide full native language support, allowing users the full power of the SunView desktop for use with their native language.

Figure 6-1 *German and French Characters in SunView 1 Desktop*

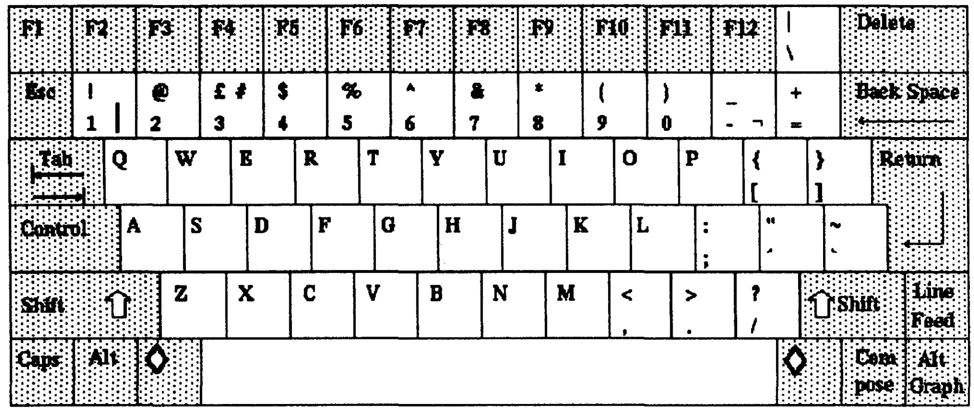


Native Language Keyboards

In 4.1, the Type 4 keyboard generates ISO Latin 1 characters. Sun also provides Type 4 keyboards with key layouts for use in a number of countries, including: Belgium, Canada, Denmark, Germany, Italy, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland (French), Switzerland (German), the United Kingdom, and the U.S.A.

Each native-language keyboard supplies the proper layout and key encodings for a specific country's language. For instance, here is the layout for the United Kingdom native-language keyboard:

Figure 6-2 *United Kingdom keyboard layout*



Appendix B shows the layouts for keyboards that are currently available.

At boot time, the system executes the `loadkeys` command which configures the key-to-character-code mappings for the keyboard. The user may run `loadkeys` at any time to update the key mappings (when switching keyboards, for instance). Applications that read from the keyboard directly, that is, without translation, must perform their own key mappings. For more information about key mappings, refer to the *SunView System Programmer's Guide*.

Alternate Key Mappings

It is also possible to generate alternate key mappings for specific uses. Existing key mappings can be found in the directory `/usr/share/lib/keytables`, which can be copied and modified using a text editor. The new key mapping thus created can be installed and brought up automatically by placing the command

`loadkeys` *keymap*

in the user's `.login` or `.profile`. Refer to `loadkeys`(8), `kb`(4M), and `keytables`(5) for details.

The Compose Key

Characters that do not appear in the layout of a given keyboard may still be entered by way of the [Compose] key. Such characters are typically composite characters that include diacritical marks. To indicate a composite character, first press the [Compose] key. Next, press the key for the desired diacritical mark, and then the key for the desired alphabetical character, or vice versa. For a complete listing of composite key sequences, refer to Appendix C, *Compose Key and Floating Accent Key Sequences*.

Floating Accent Keys

On some keyboards, certain keys appear with an empty box (□) underneath the diacritical mark. These are referred to as *floating accent* keys. When used, they allow you to type in a composite character without using the [Compose] key. The floating accent key must be typed first, followed by the key for the character to be accented.

Line Printers

4.1 support for native language printing includes:

- Transmission of 8-bit characters by `lpr`. The serial line must also be 8-bit clean, and printer must support the ISO Latin 1 character set for printing to take place. Otherwise, the output must be filtered before printing can take place.
- PostScript†-based printing using TranScript®, an optional software package that provides PostScript-based printing on Sun's LaserWriter® printer products.

Networking

The TCP/IP and UDP protocols provide 8-bit clean datapaths for interprocess and network communication, but this is no guarantee that applications using these protocols will not interpret the 8th bit. The RPC-based services provided with release 4.1 are 8-bit clean. Internet-protocol services in 4.1 are also 8-bit clean. They will handle 8-bit code sets in addition to ISO Latin 1, provided that those code sets also incorporate the encodings for printable ASCII characters.

Mailers

The electronic mail applications, `/usr/bin/mail` and `/usr/ucb/mail` (`Mail`), can handle 8-bit text. However, they are not designed to transfer binary data that does not conform to the text model; files that do not include [Return] characters within a normal line-length range, or that are not null-terminated, may not get through.

The mail message-delivery server in 4.1, `sendmail`, can also handle 8-bit text. However, not all implementations of `sendmail` are 8-bit clean. Versions released prior to 4.0, or those supplied with other operating systems, are known to strip the 8-th bit from text included in messages.

File Transfer and Sharing

Since NFS does not interpret the file's content, text files with 8-bit characters can be shared across systems. Also, in 4.1, pathnames can contain 8-bit characters. However, servers running releases prior to 4.0 may have difficulty with these filenames.

`uucp`(1) can handle 8-bit text, but not binaries (unless they are encoded using `uuencode`(1)).

When used in binary mode, `ftp` can transfer 8-bit text files with no problems. There may be problems when trying to transfer 8-bit text using `ftp` in `ASCII` mode.

Terminal Emulation

When used with a serial line operating in 8-bit, noparity mode, `tip` can pass 8-bit characters to a terminal capable of displaying them. `telnet` is officially a 7-bit protocol, however it too has been rendered 8-bit clean in 4.1.

[2] †PostScript™ is a trademark of Adobe Systems Incorporated.

Other Networking Services

comsat the server for the mail notifier biff is 8-bit clean, as are finger(1), and talk(1).

rsh(1), the remote shell, is 8-bit clean, as is rlogin, but will only function as such if the remote host is also running an 8-bit clean shell.

Modems

When used with 8-bit data, modems should be set to 8-bit noparity mode.

The Announcement (Locale) Mechanism

The key concept for application programs is that of a program's locale. The locale is an explicit model and definition of a native-language environment. The notion of a locale is explicitly defined and included in the library definitions of the proposed ANSI C Language standard.

A program's locale defines items such as its code set (typically a subset of ISO Latin 1), date and time formatting conventions, monetary and decimal formatting conventions, and collation order.

The locale consists of a number of categories for which there are language-dependent formatting or other specifications.

In Release 4.1, these categories take the form of subdirectories in the localization-database file hierarchy. The set of files corresponding to a given locale (represented by a file in each category's subdirectory) is referred to as a *localization*.

The localization subdirectories are as follows:

LC_CTYPE
: For controlling the behavior of character-handling routines and multi-byte character functions.

LC_TIME
: Date-time formats.

LC_MONETARY
: Monetary formats.

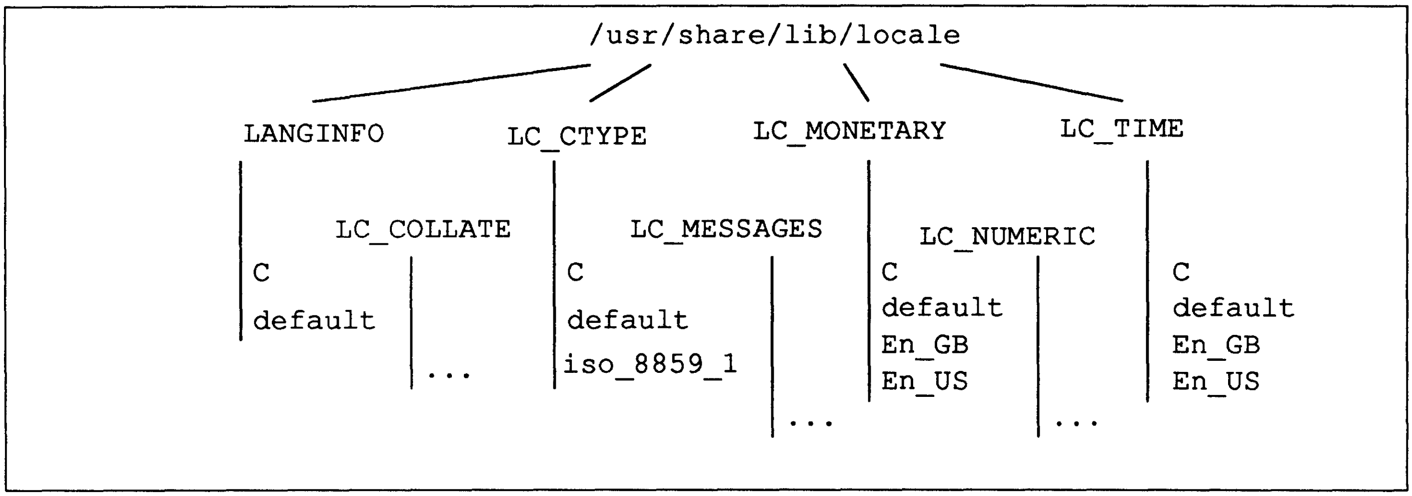
LC_NUMERIC
: Numeric formats and decimal-point characters.

LC_COLLATE
: Character (case) conversions and string collation tables.

LC_MESSAGES
: Message catalogs.

LANGINFO
: Used by nl_langinfo() to display information about the locale.

Figure 6-3 *Structure of a Localization Database*



Each of these directories has a corresponding environment variable of the same name. A specification for each category can be obtained or altered by calling `setlocale()` and specifying the category and value. For example:

```
setlocale(LC_NUMERIC, "En_GB");
```

sets the format for numeric values to that for Great Britain.

In addition to identifying the code set, the `LC_CTYPE` can be used to indicate the user's overall native-language environment. In other words, in the absence of a specific call to `setlocale` for a given category, the system can be instructed to use the value of `LC_CTYPE` for all categories.

When called as shown:

```
setlocale(LC_ALL, "")
```

`setlocale()` attempts to use the filename indicated in the `LC_CTYPE` variable for each category. If `LC_CTYPE` is empty or invalid (no corresponding file), `setlocale()` tries the value of the `LANG`, environment variable, and then that of `LC_DEFAULT`. If none of these apply, `setlocale()` uses the `default` file in each directory. This file is typically a symbolic link to another file within the subdirectory. The standard default in 4.1 is the ''C'' locale.

In accordance with the proposed POSIX 1003.1 standard and XPG2, 4.1 also provides the `LANG` and `NLSPATH` environment variables to announce the run-time locale requirements, and to indicate the directory search path for message catalogs, respectively.

The environment variable `LANG` can be used to identify the locale. However, the value for `LC_CTYPE` takes precedence over `LANG`. A recognized value for `LANG` takes the following form, which specifies the native language, and further qualifies it if necessary with territory and codeset specifications:

```
language[_territory[.codeset]]
```

which is used to name the locale information file in each localization category.

In practice, the *territory* and *codeset* fields are unnecessary with the default environments shipped with 4.1. In 4.1 this syntax is not used to select supersets of an individual language. Instead the programmer should use the individual categories mentioned above, with the same territory.codeset structure as necessary.

The `NLSPATH` (environment) variable determines the search path for the localization database(s). The default value for `NLSPATH` is `/usr/share/lib/locale`.

6.2. Using the Internationalized Desktop

Using `LC_CTYPE` to switch display locales does not affect keyboard input. Thus, if you had French and German locales available for your application, used a German keyboard, and switched locales from German to French, the keyboard would still transmit the labeled German characters. To enter French characters, you would either have to use the [Compose] key, or switch physical keyboards and issue a `loadkeys` command. (Or, if so inclined, you could set the dip switches inside the keyboard, but the key caps would then be inaccurate.)

Sharing Data between Applications

Many existing 3rd-party applications are based on the 7-bit ASCII codeset. Since applications that do not perform an explicit `setlocale()` call operate with the "C" environment, they will operate in the 4.1 without problems. However, they may have difficulty with 8-bit character input from files or devices. Other applications may use different codesets. To deal with the limitations of such applications, consider:

- Does the application use a different codeset?

  If so, then, it may be possible to create a filter to map text between the application and the system, or between applications.

- Does the application use the ISO Latin 1 codeset?

  If not, then long in the long-term it should be changed to do so. In the short term, you may be able to generate a character classification table for the existing code set (such as the IBM PC international codeset).

- Can the application use 4.1 functionality? By default, the window system and screen fonts assume the codeset to be ISO Latin 1. To use another you would have to supply appropriate fonts and a new character classification table. But note that, aside from editors, system utilities do not ordinarily destroy data contained in files, they simply misinterpret it.

Applications that use ISO Latin 1 as their code set will operate cleanly under 4.1. Since 4.1 is 8-bit clean, applications based on other codesets should also operate on their own files without problems. However, attempts to mix and match files may have unexpected results.

Sharing Data between 4.1 Host Systems

Data sharing and interprocess communication between host systems running Release 4.1 is completely transparent.

Sharing Data with Other SunOS Operating System Hosts

When attempting to share data between 4.1 hosts and host systems running earlier versions of the operating system (or other operating systems such as UNIX System V or BSD):

- 4.1 Host as File Server, with Non-4.1 Client

  In this configuration, you cannot assume that the applications running on the client are 8-bit clean. Thus, 8-bit text files used by the client may create problems. If the client is running 4.0, the Bourne shell is 8-bit clean. However, most utilities are not.

- 4.1 Client, Non-4.1 Server

  In this configuration, the file-access capabilities of the server allow 8-bit text files to be accessed by client applications. However, utility programs running on the server itself may have trouble with 8-bit text. If the server is running 4.0, 8-bit characters in filenames are allowed. This may not be true for other systems.

6.3. Creating and Installing a Native Language Environment (Locale)

Now that you've been introduced to the native language support features, it is time to discuss how to create a locale. Creating a locale involves:

- Selecting a name for the new locale
- Creating and installing a character classification and conversion table, and a string collation table
- Creating and installing formats for dates and times, monetary values, and numeric values
- Creating a database for native-languages messages for use by an application.

These topics are discussed in the following sections.

Building a Classification and Conversion Table: `chrtbl`

The `chrtbl(8)` command is used to create a table that contains the character classification and case conversion tables for a code set. `chrtbl` takes as input a specification file, and produces a classification and conversion data file that is in proper format for use within the `LC_CTYPE` localization category.

The input file for the ISO Latin 1 code set is shown below. Note that characters are specified by their hexadecimal (or octal) values. The - character is used to indicate a range of values, while \ is used to continue across input lines. Lines that begin with a # are treated as comments. The relationship between lower and upper case letters is expressed as bracketed ordered pairs, with the first element being lower-case.

```
# ISO Latin 1 Code Set definition
chrclass      iso_8859_1
model         euc 1,1,1
isupper       0x41-0x5a 0xc0-0xd6 0xd8-0xde
islower       0x61-0x7a 0xdf 0xe0-0xf6 0xf8-0xff
isdigit       0x30-0x39
isspace       0x20 0x09-0x0d 0xa0
ispunct       0x21-0x2f 0x3a-0x40 0x5b-0x60 0x7b-0x7e \
              0xa1-0xbf 0xd7 0xf7
iscntrl       0x0-0x1f 0x7f
isblank       0x20 0xa0
isxdigit      0x30-0x39 0x61-0x66 0x41-0x46
ul      <0x41 0x61> <0x42 0x62> <0x43 0x63> <0x44 0x64> \
        <0x45 0x65> <0x46 0x66> <0x47 0x67> <0x48 0x68> \
        <0x49 0x69> <0x4a 0x6a> <0x4b 0x6b> <0x4c 0x6c> \
        <0x4d 0x6d> <0x4e 0x6e> <0x4f 0x6f> <0x50 0x70> \
        <0x51 0x71> <0x52 0x72> <0x53 0x73> <0x54 0x74> \
        <0x55 0x75> <0x56 0x76> <0x57 0x77> <0x58 0x78> \
        <0x59 0x79> <0x5a 0x7a> <0xc0 0xe0> <0xc1 0xe1> \
        <0xc2 0xe2> <0xc3 0xe3> <0xc4 0xe4> <0xc5 0xe5> \
        <0xc6 0xe6> <0xc7 0xe7> <0xc8 0xe8> <0xc9 0xe9> \
        <0xca 0xea> <0xcb 0xeb> <0xcc 0xec> <0xcd 0xed> \
        <0xce 0xee> <0xcf 0xef> <0xd0 0xf0> <0xd1 0xf1> \
        <0xd2 0xf2> <0xd3 0xf3> <0xd4 0xf4> <0xd5 0xf5> \
        <0xd6 0xf6> <0xd8 0xf8> <0xd9 0xf9> <0xda 0xfa> \
        <0xdb 0xfb> <0xdc 0xfc> <0xdd 0xfd> <0xde 0xfe>
```

The `chrclass` heading gives the name of the code set. The value for this heading is used by `chrtbl` as the basename for the output file. The optional `model` heading gives a description of the rules for a particular codeset. These rules will affect the way in which the multi-byte functions defined in `mblen`(3) operate. If the model field is selected and has the correct syntax it will crate another output file with a filename based on the `chrclass` heading, with a `.ci` suffix added. If the `model` heading is not used then it is assumed that the code-set being defined is a single byte codeset. The `ul` heading indicates that the upper-to-lower case mappings follow. The other headings indicate which characters are to be recognized by the various character-classification routines.

To compile the classification table, use a command of the form:

```
chrtbl iso_8859_1
```

and then install the table in the `LC_CTYPE` directory of the locale database, as described under *Installing a Locale*, below.

It is possible to make variants of a classification table by making small adjustments to an ISO code-set definition. This may be to show small differences of operation in differing countries. For example if you wished to make the French version of 8859/1 invalidate upper case accented letters, this could be achieved by editing the basic 8859/1 table, marking them as invalid and creating a new character set table for a French locale.

It should be noted that some issues of conversion are only handled by the collation facility. The conversions allowed by this table are only single byte to single byte. For example this table will not support the conversion of the German β character to the string "ss".

Building a String Collation Table: `colldef(8)`

The `colldef` command is used to create string collation tables used by a code set. `colldef` reads its standard input, and produces a collation table that is in proper format for use within the `LC_COLLATE` localization category.

When comparing sequences (strings) of characters, a pair of words might collate differently in different languages. The `strxfrm()` and `strcoll()` library routines allow programs to use the locale-specific collation tables for sorting strings.

A sample input file for `colldef` is shown below.

```
#
# A sample collation specification
#
order \x20;A;a;B;b;(C,c);ch;D;d;(E,\xc8,\xc9,\xca);f;...;z
substitute "\xdf" with "ss"
substitute "\xc6" with "AE"
substitute "\xe6" with "ae"
```

The `order` line gives specifies the sort order for single characters. Semicolons are used to separates primary collating elements. So, in this ordering, the string `Apple` would be sorted ahead of the string `apple`. Parentheses are used to indicate a secondary sorting, that is, groups of characters that are to be collated together in the absence of a distinguishing character to follow. Thus '`Ca`' comes before '`ca`' (as it would without the brackets), but `ca` comes before '`Ce`'. (which it wouldn't without the brackets)

The `substitution` lines define substitution rules. These are generally used during sorting, so strings such as the following:

```
schloβ
schloss
```

(in this example) will collate together.

To compile the collating table, use a command of the form:

```
cat spanish.src | colldef spanish
```

and then install the compiled table in the `LC_COLLATE` directory of the locale database.

Date and Time Formats

Different cultures and nations use a variety of conventions to record the date and time. The following table illustrates the wide variety of conventions in use around the world.

The `strftime`(3) function can be used to display the date and time in the desired format.

Table 6-2 *International Date and Time Conventions*

Language	Convention	Examples
Danish	*dd/mm/yy*	13/08/89
Finnish	*dd.mm.yyyy*	13.08.1989
French	*dd/mm/yy*	13/08/89
German	*dd.mm.yy*	13.08.89
Italian	*dd.mm.yy*	13.08.89
Norwegian	*dd.mm.yy*	13.08.89
Spanish	*dd-mm-yy*	13-08-89
Swedish	*yyyy-mm-dd*	1989-08-13
United Kingdom	*dd/mm/yy*	13/08/89
United States	*mm-dd-yy*	08-13-89
French Canadian	*yyyy-mm-dd*	1989-08-13
English Canadian	*yyyy-mm-dd*	1989-08-13

The simplest way to create a date and time table for the `LC_TIME` category is to follow the format given in the file:
`/usr/share/lib/locale/LC_TIME/C`:

```
Jan
Feb
Mar
...
Dec
January
February
March
...
December
Sun
Mon
...
Sat
Sunday
Monday
...
Saturday
%H:%M:%S
%m/%d/%y
%a %b %e %T %Z %Y
AM
PM
%A, %B %e, %Y
```

The first twelve lines indicate the short forms of the months of the year. The following twelve give the long forms. The next seven give the short form of the days of the week. The following seven give the long forms. The next lines give various date and time formats using the field descriptors described in ctime(3V):

`%H:%M:%S`	Short form of local time
`%m/%d/%y`	Short form of local date
`%a %b %e %T %Z %Y`	Local short form for date and time.
`AM`	ante meridiem notation
`PM`	post meridiem notation
`%A, %B %e, %Y`	local long form for date and time

The text of these last lines can be altered as to punctuation, order and content according to local custom.

Once the new date and time format file has been completed, you can install the file in the `LC_TIME` directory of the locale database.

Decimal Units

There are a variety of formatting conventions for decimal units as well, as the following table shows:

Table 6-3 *International Decimal Formatting Conventions*

Language	Examples
Danish	1.234.567,89
Finnish	1.234.567,89
French	1.234.567,89
German	1 234 567,89
Italian	1.234.567,89
Norwegian	1.234.567,89
Spanish	1.234.567,89
Swedish	1.234.567,89
United Kingdom	1,234,567.89
United States	1,234,567.89
French Canadian	1 234 567,89
English Canadian	1 234 567,89

You can use the `fscanf()` (refer to `scanf`(3C)) routine to accept input of decimal amounts. `fscanf()` has been enhanced in 4.1 to accommodate different input formats. Currently `scanf()` will not understand the space as a valid input separator, but the space can be used on output (German uses both modes).

To create a numeric format specification for the `LC_NUMERIC` category, follow the format given in the file
`/usr/share/lib/locale/LC_NUMERIC/En_US`:

```
.
,
3
```

The first line of this file contains the radix character. The second line contains the thousands-separator, and the third line gives the number of digits for grouping purposes. If the last two lines are empty, grouping (by thousands) is not done.

Once the numeric format file has been completed, you can install it in the `LC_NUMERIC` directory of the locale database.

Monetary Formats

There are many different formats for monetary figures, as the table below illustrates.

Table 6-4 *International Monetary Formatting Conventions*

Language	Unit of Currency	Example
Danish	Kroner(kr)	`kr.1.234,56`
Finnish	Markka(mk)	`1.234 mk`
French	Franc(F)	`F1.234,56`
German	Deutschemark(DM)	`1,234.56DM`
Italian	Lira(L)	`L1.234,56`
Norwegian	Krone(kr)	`kr 1.234,56`
Spanish	Peseta(Pts)	`1.234,56Pts`
Swedish	Krona(Kr)	`1234.56KR`
United Kingdom†	Pound(#)	`#1,234.56`
United States	Dollar($)	`$1,234.56`
English Canadian	Dollar($)	`$1 234.56`
French Canadian	Dollar($)	`1 234.56$`
† The symbol # represents the pound-sterling symbol		

The `localeconv`(3) function is used to obtain currency formats. It uses the formatting conventions of the current locale to set the components of an object with type `struct lconv` to the appropriate values, and returns a pointer to the filled-in object.

To create a currency format specification for the LC_MONETARY category, follow the format given in the file
/usr/share/lib/locale/LC_MONETARY/En_US:

```
USDO
$
.
,
3
+
-

2
y
n
y
n
1
0
```

This file consists of exactly fifteen lines, each of which contains specific information about the monetary format:

Line 1. International Currency Symbol (string)
This is the currency symbol for the locale. The first three characters contain the alphabetical code for the symbol as specified in ISO 4217, *Codes for the Representation of Currency and Funds.* The fourth character, which must also be the last character on the line, is the character used to separate the currency symbol from the monetary quantity. For example:

```
ITL.
```

would be the correct specification for Italy. ITL refers to the standard code for the currency, and the period separates the code from the amount. Thus, the string ITL.123,000 would represent 123,000 Lire.

Line 2. Local Currency Symbol (string)
This is the local version of the currency symbol, such as the $ dollar-sign used in the United States.

Line 3. Monetary Decimal Point (string)
This is the radix character used to format monetary quantities. It separates the unit quantity from the decimal fraction parts. If this is empty, it means by default the decimal parts are not printed (such as in Italy, where fractions of Lire are not printed).

Line 4. Monetary Thousands Separator (string)
This is the string used to separate digits that are grouped together. It is usually a comma or period, and most often groups together thousands units (3 digits). If this line is blank, no grouping character is used.

Line 5. Monetary Grouping Specification (string)
This line gives the size of a group of digits. It is often used only for

separation after the thousands digit, but may be use in higher groupings as well. For example:

`\3`	separates after thousands digit only:	`7654,321`
`3\3`	separates after each group of 3 digits:	`7,654,321`

If this line is empty, no grouping is done.

Line 6. The Positive Sign (string)
The symbol used to represent a positive value. It is normally empty, but may sometimes contain a symbol such as the plus sign (+). If this line is empty, no positive sign is required.

Line 7. The Negative Sign (string)
The symbol used to represent a negative value. Usually set to the minus sign (-).

Line 8. International Fractional Digits Count (character)
This is the integer number of digits required after the decimal point in the international monetary representation. This does not affect the local representation. For instance, the value 2 would produce:

```
NLG 1.234.56
```

for Dutch Guilders. If this line is empty, fractional digits are not represented.

Line 9. Local Fractional Digits Count (character)
This is the integer number of digits required after the decimal point in the local monetary representation. The value 3 would produce:

```
$1,234.560
```

for U.S. Dollars (obviously not the standard presentation form in this case). If this line is empty, fractional digits are not represented.

Line 10. Position of Currency Symbol when Positive (character)
This is a boolean value that indicates whether the currency symbol comes to the left or right of a positive (nonnegative) value. An `y` (or `Y`, `t`, or `T`) means that the symbol appears to the left, an `n` (or `N`, `f`, or `F`), to the right. If this line is empty, it is taken as `f`.

Line 11. Space Separation of Currency Symbol for Positive Values (character)
If this line contains an `y` (or `Y`, `t`, or `T`), the currency symbol is separated by a space from the positive monetary value. Otherwise the symbol is not separated from the value.

Line 12. Position of Currency Symbol when Negative (character)
This is a boolean value that indicates whether the currency symbol comes to the left or right of a negative value. A `y` (or `Y`, `t`, or `T`), means that the symbol appears to the left, an `n` (or `N`, `f`, or `F`), to the right. If this line is empty, it is taken as `0`.

Line 13. Space Separation of Currency Symbol for Negative Values (character)
If this line contains a `y`, (or `Y`, `t`, or `T`), the currency symbol is separated by a space from the negative monetary value. Otherwise the symbol is not

separated from the value.

Line 14. Position of Positive Sign (character)
This is a numeric value in the range 0-4, representing the position of the positive sign with respect to the monetary value, as follows:

0	Parentheses surround the currency symbol
1	The sign string precedes the quantity and currency symbol
2	The sign string succeeds the quantity and currency symbol
3	The sign string immediately precedes the currency symbol
4	The sign string immediately succeeds the currency symbol

Line 15. Position of Negative Sign (character)
This is a numeric value in the range 0-4, representing the position of the positive sign with respect to the monetary value, as follows:

0	Parentheses surround the currency symbol
1	The sign string precedes the quantity and currency symbol
2	The sign string succeeds the quantity and currency symbol
3	The sign string immediately precedes the currency symbol
4	The sign string immediately succeeds the currency symbol

Message Catalogs

Message catalogs are files of message strings, separated from an application, with an indexed internal structure. They are accessed by file name. The `gencat(1)` utility is used to create a message catalog from the message text source file.

Individual messages are indexed by `msg_id` within the catalog. Optionally, message catalogs can also be divided into one or more sets of message, which are indexed by `set_id`. Given these identifiers, accessing the appropriate message is a simple table lookup.

Unlike the other categories in the locale database, the `LC_MESSAGES` directory contains subdirectories for each locale. Each individual message catalogue typically resides within each subdirectory associated with every available locale (language) of messages for an application.

To build a message catalog for a given application and locale, first extract the message strings from the source file. With this release of SunOS, there are no tools supplied to automate this process.

The 4.1 C library allows you to make reference to a message string through the functions `catgets(3)` and `catopen(3)`. In addition 4.1 supplies the `gettext(3)`, and `textdomain(3)` functions for the same purpose. Both sets of functions perform the same tasks, although it is not recommended to mix both sets of calls in the same application.

For an X/OPEN compliant application, run the source message file through `gencat(1)`. This will produce a binary message file in the current working directory that can later be moved to the correct installation directory.

If the message text is built for use with `gettext()`, you may use the `installtxt(1)` to build it, and as with `gencat` you can copy the binary into the locale database for run-time loading.

Installing a Locale

Once the various files for the desired categories have been created, you can install them in the default locale database (directory tree), provided that you can become the super-user on the system. The pathname for this location is:
`/usr/share/lib/locale`
If you wish to install a per-workstation private version of the same database, you may install the files under:
`/etc/locale`
Which is always searched first by the `setlocale()` function.

6.4. Developing an Internationalized Application

Creating internationalized application programs is not difficult, but it does require knowledge of some specific programming techniques. If the need for internationalization is considered in the application's design, the development process can be quite straightforward. Techniques for dealing with the various categories governed by the current locale are described in this section.

Programmers building internationalized applications may also be interested in several other references. The *Draft Proposed National Standard for Information Systems—Programming Language C* explains the entire C language interface, and is available from the:

> X3 Secretariat
> Computer and Business Equipment Manufacturers Association
> 311 First Street, N.W., Suite 500
> Washington, DC 20001.

The *X/OPEN Portability Guide* volumes 2 and 3, explains the X/OPEN requirements for internationalization; it is written by the X/OPEN Company, Ltd. and is printed and published by:

> Prentice Hall
> Englewood Cliffs, NJ 07632

Note that the C compiler does not support 8-bit characters in object names (that is, names of routines, variables, and so forth), although it does allow you to initialize 8-bit data in strings. Certain 8-bit characters are treated specially by `cpp`, and so their use is not recommended in names of defined constants.

Overview

This section discusses the following considerations when designing an application, and provides short programming examples of the best ways to structure software.

- Acquiring the native-language environment using `setlocale()`
- Handling of alternate alphabets and character sets
- Date and Time Formats
- Numeric Formats
- Monetary Formats
- File Names

- Sorting and Collation Orders
- Native Language Messages
- Other Considerations

8-Bit Character Support Routines

Release 4.1 provides the following library routines for 8-bit character support.

Table 6-5 *Internationalized Routines*

Internationalized Routines	
Routine	**Description**
Locale	
`localdtconv()`	Returns date and time format for locale
`localeconv()`	Returns numeric and monetary formats
`setlocale()`	Set locale or locale category
Date/Time	
`strftime()`	Convert date and time to string
`strptime()`	Convert string to date and time
Character Handling	
`isalnum()`	Character classifications
`isalpha()`	
`isascii()`	
`iscntrl()`	
`isdigit()`	
`isgraph()`	
`islower()`	
`isprint()`	
`ispunct()`	
`isspace()`	
`isupper()`	
`isxdigit()`	
`toascii()`	Character conversions
`tolower()`	
`toupper()`	
String Handling	
`atof()`	Convert string to number
`ecvt()`	Convert number to string
`fcvt()`	
`gcvt()`	
`regexp`(3)†	Regular-expression routines
`strcoll()`	Collate two strings
`strtod()`	Convert string to number
`strxfrm()`	Transform string
Formatted Output	
`fprintf()`	Print formatted string
`printf()`	
`sprintf()`	
`nl_fprintf()`	Print formatted string (XPG2 version)

Table 6-5 *Internationalized Routines— Continued*

Internationalized Routines	
Routine	**Description**
nl_printf()	
nl_sprintf()	
Formatted Input	
scanf()	Accept formatted input
fscanf()	
sscanf()	
nl_scanf()	Accept formatted input (XPG2 version)
nl_fscanf()	
nl_sscanf()	
Messaging	
catgets()	X/Open Messaging function
catgetmsg()	X/Open Messaging function
catopen()	X/Open Messaging function
catclose()	X/Open Messaging function
gettext()	Messaging function
textdomain()	Messaging function
nl_langinfo()	Print native-language database info
Multi-Byte Characters ‡	
mblen()	Get length of multi-byte string
mbtowc()	Multi-byte to wide character
wctomb()	Wide character to multi-byte character
mbstowcs()	Multi-byte string to wide character string
wcstombs()	Wide character string to multi-byte string
† regexp(3) routines are 8-bit clean only. They do not handle POSIX regular expressions.	
‡ These routines support a number of multi-byte code sets, including: EUC, ISO 2022, and XEROX XCCS.®.	

The SunView 1 input and display routines also support 8-bit characters.

Acquiring the Locale: `setlocale()`

To conform with the ANSI C language standard, all processes are initialized to use the "C" (ASCII) native-language environment. Therefore, a program must make an explicit call to `setlocale()` in order to use the locale specified in its environment. A call of the form:

```
setlocale(LC_ALL, "");
```

is typically used to set all locale categories to those in the environment.

Applications may allow users to modify one or more locale categories, or to switch locales entirely, by calling `setlocale()`.

Handling Alphabets and Character Sets

Internationalized applications eliminate codeset dependencies. Self-developed programming techniques that introduce dependencies on the ASCII codeset must be converted to a more portable form for an application to successfully handle varying code sets. For instance, the example below shows a hard-coded test based on ASCII, which should be replaced with `isprint()`, one of the standard character-range test routines listed above. This program will fail to correctly recognize some ISO Latin 1 characters that are printable when run in a locale other than "C."

```
/* Poor practice: Codeset Assumed to Be ASCII   */
main()
{
    int c;
    if (c<=037||c>=0177)
    printf("This character cannot be printed\n");
    else
    printf("This character is %c\n",c);
}
```

Handling Date and Time Formats

As mentioned earlier, `strftime()` can be used to display the date and time in whatever form the current locale specifies. `strftime()`, `strptime()`, and `localdtconv`, are other functions that handle locale-dependent time formats (see `ctime`(3V). The synopsis of `strftime()` is:

```
#include <time.h>

size_t strftime(s, maxsize, format, timeptr);
char *s;
size_t maxsize;
char *format;
struct tm *timeptr;
```

where `s` is a pointer to a string in which to store the formatted time, `maxsize` is the maximum number of bytes that will be placed in `s`, `format` is a string giving the format to display, and `timeptr` is a pointer to a tm struct as returned by `localtime()`.

For example, the function below displays the time correctly in a number of different locales:

```
#include <time.h>
#include <sys/types.h>
#include <locale.h>
#define MAXLEN  80
int  strftime();  /*Returns date/time according to locale */
char     buff[MAXLEN];
struct   tm *timeptr;
time_t   clock;
int  count;
main()
{
     setlocale(LC_TIME,"");
         clock = time(0);
         timeptr = localtime(&clock);
         count=strftime(buff,MAXLEN,"%x %X",timeptr);
         printf("Todays Date/Time Is: %s\n",buff);
}
```

Handling Numeric Formats
`scanf()`

It is possible to use the `scanf` style functions to input data based on language dependent grammar, or order (see `scanf`(3V)). The trick here is to be able to vary the format string without the need to change the (hard-coded) argument lists in your program code. The format string can be extracted and can be defined in a locale dependent manner.

```
int fscanf(stream, format [, pointer]... )
FILE *stream;
char *format;
```

`fscanf()` reads input from the stream pointed to by `stream`; the string pointed to by `format` specifies the admissible input sequences and an (optional) order in which they are to be converted for assignment, for example, the call:

```
char input_string[40] = "dirty water";
char adjective[20], noun[20];
sscanf(input_string,"%1$s%2$%s",adjective,noun);
```

would place "dirty" in the string `adjective`, and "water" in the string `noun` Now, in German it may be required to reverse the noun and adjective, in which case we would only have to change the (possibly extracted) string in the above example, as follows:

```
sscanf(input_string,"%2$s%1$s",adjective,noun);
```

`localeconv`

`localeconv()` returns a pointer to the `lconv` structure, which contains data for formatting numeric and monetary amounts. This can be useful in conjunction with conversion routines such as `atof()`, for converting input strings into actual numeric values.

The components of the `lconv` structure are given in `<locale.h>` as shown:

```
/*
 * Numeric and monetary conversion information.
 */
struct lconv {
    char    *decimal_point;     /* decimal point character */
    char    *thousands_sep;     /* thousands separator character */
    char    *grouping;          /* grouping of digits */
    char    *int_curr_symbol;   /* international currency symbol */
    char    *currency_symbol;   /* local currency symbol */
    char    *mon_decimal_point; /* monetary decimal point character */
    char    *mon_thousands_sep; /* monetary thousands separator */
    char    *mon_grouping;      /* monetary grouping of digits */
    char    *positive_sign;     /* monetary credit symbol */
    char    *negative_sign;     /* monetary debit symbol */
    char    int_frac_digits;    /* intl monetary number of fractional digits */
    char    frac_digits;        /* monetary number of fractional digits */
    char    p_cs_precedes;      /* true if currency symbol precedes credit */
    char    p_sep_by_space;     /* true if space separates c.s.  from credit */
    char    n_cs_precedes;      /* true if currency symbol precedes debit */
    char    n_sep_by_space;     /* true if space separates c.s.  from debit */
    char    p_sign_posn;        /* position of sign for credit */
    char    n_sign_posn;        /* position of sign for debit */
};
```

Alternative input routines are also provided. The `scanf()` and `sscanf()` functions can be used to read from the standard input stream, or from a character string, respectively. For compatibility with XPG2, the routines `nl_scanf()`, `nl_sscanf()` and `nl_fscanf()` are also provided. However their use is not recommended since their functionality has been completely subsumed by the `scanf()` routines as specified in XPG3.

`printf()`

It is possible to use the `printf` style functions to output data based on language dependent grammar, or order (see `scanf`(3V)). The trick here (as with `scanf()`) is to be able to vary the format string without the need to change the (hard-coded) argument lists in your program code. The format string can be extracted and can be defined in a locale dependent manner.

```
int fprintf(stream, format [, pointer]...)
FILE *stream;
char *format;
```

`fprintf()` writes output to the stream pointed to by `stream`; the `format` string specifies how subsequent arguments are converted for output. For instance in American usage:

```
fprintf(stream,"%s, %s %d, %d:%.2d\n",day,month,date,hour,minute);
```

might produce:

```
Sunday, July 3,10:02
```

Whereas for German usage, the format string could be replaced:

```
fprintf(stream, "%1$s, %3$d.%2$s,%4$d:%5$.2d\n",
    day, month, date, hour, minute);
```

to produce:

```
Sonntag, 3.Juli,10:02
```

Alternative output routines are also provided. The `printf()` and `sprintf()` functions can be used to output to the standard output stream, or to a character string, respectively. For compatibility with XPG2, the routines `nl_printf()`, `nl_sprintf()` and `nl_fprintf()` are also provided. However, since their functionality has been subsumed by the `printf()` family in XPG3, their use is not recommended.

Handling Monetary Formats

The table below illustrates the rules that might be used by three countries. The table that follows shows respective values for the structure that would returned by `localeconv()`, once the appropriate locales have been created and installed.†

Table 6-6 *More Sample Monetary Formats*

Country	Positive Format	Negative Format	International Format
Italy	`L.1.234`	`-L.1.234`	`ITL.1.234`
Netherlands	`F 1.234,56`	`F -1.234,56`	`NLG 1.234,56`
Norway	`kr1.234,56`	`kr1.234,56-`	`NOK 1.234,56`
Switzerland	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`

[2] †These locales are not supplied in Release 4.1.

Table 6-7 *Values of the Structure Returned by* `localeconv()`

Field	Italy	Nether-lands	Norway	Switzer-land
`int_curr_symbol`	"ITL."	"NLG "	"NOK ".	"CHF "
`currency_symbol`	"L."	"F"	"kr"	"SFrs."
`mon_decimal_point`	""	","	","	"."
`mon_thousands_sep`	"."	"."	"."	","
`mon_grouping`	"\3"	"\3"	"\3"	"\3"
`positive_sign`	""	""	""	""
`negative_sign`	"-"	"-"	"-"	"C"
`int_frac_digits`	0	2	2	2
`frac_digits`	0	2	2	2
`p_cs_precedes`	1	1	1	1
`p_sep_by_space`	0	1	0	0
`n_cs_precedes`	1	1	1	1
`n_sep_by_space`	0	1	0	0
`p_sign_posn`	1	1	1	1
`n_sign_posn`	1	4	2	2

There are no currently accepted international standard routines to control the input of formatted monetary information. Programmers should use `localeconv()` in conjunction with `fscanf()` or `read()` to construct their own input routines. Similarly, there are no currently accepted international standard routines to control the output of formatted monetary information. Programmers should use the `localeconv()` and `fprintf()` to construct their own.

Handling File Names

Release 4.1 allows for any ISO 8859/1 character to be a valid character within a file name except for the backslash (\), SPACE, slash / and NULL characters. It is assumed the normal conventions for filenames will be applied to (e.g. The `.c` suffixes).

Sorting, Collation and Conversion

The correct sorting of an alphabetic list, or *collation* across European languages is a much more difficult problem than it appears at first glance. Many factors affect collation order.

Often, accented characters and unaccented characters should sort alike. Upper case and lower case characters should sort alike. Accented characters usually follow unaccented characters. However, there are many exceptions to this rule. Some accented characters sort as a unique letter; some double characters sort as a single character. Many more complex rules apply.

SunOS provides two functions for string comparison: `strcoll()` and `strxfrm()`. Both of these reference the collation information in the program's language locale, (category `LC_COLLATE`). The collation sequence table in the locale can, in turn, be accessed or initialized from the command line with the `colldef` and `chartbl` commands. SunOS 4.1 provides no collation tables by default in the standard software distribution. Developers requiring collation tables must construct their own.

The `strcoll()` function compares the string pointed to by its first argument with the string pointed to by its second, interpreted with respect to the `LC_COLLATE` category of the current locale. The sign of a non-zero value returned is determined by the relative ordering within the current collating sequence of the first pair of characters which differ.

The `strxfrm(s1,s2,n)` function transforms the string pointed to by `s2` and places the resulting function into the array pointed to by `s1`. The transformation is such that two transformed strings can be ordered by `strcmp()`.

Native-Language Messages

Release 4.1 provides several alternative solutions to the problem of how to create message structures which can be easily written, translated, and correctly accessed at run-time depending upon the locale of the program. Messages are stored in *message catalogs*, files containing messages which are indexed and accessible by `msg_id`.

Because the contents of the message catalog are separate from the application's code, a message catalog for the current locale can be selected or altered at run-time without altering the code itself.

Library Routines for Accessing Message Catalogs

Message catalogs are opened by calling the routine `catopen()`, which locates the identified message catalog accord to the search and naming rules in the environment variable `NLSPATH`. To illustrate:

```
#include <nl_types.h>
nl_catd catd = catopen("catalog_name",0);
```

will return a catalog descriptor, `nl_catd` which is then used in calls to `catgets()` to identify the message catalog. Message catalogs are closed with the routine `catclose()`.

The routine `catgets()` uses a message identifier, `msg_id`, to extract from the numbered message set identified by `set_id`, within the catalog referred to the by the catalog identifier, `catd`:

```
char *catgets(catd,set_id,msg_id,string);
```

The small program below illustrates the use of all the routines. It retrieves the first message of the second set of catalog messages in the file `catalog_name`. If the call fails, the program displays the string: '`Not successful text`'.

```
#include<stdio.h>
#include <nl_types.h>
#include <locale.h>

#define SET_NUMBER 2
#define MESSAGE_NUMBER 1

main()
{
    nl_catd catd;

    setlocale(LC_MESSAGES,"");
    catd = catopen("catalog_name",0);
    printf("%s\n",catgets(catd,SET_NUMBER,MESSAGE_NUMBER,
              "default text"));
    catclose(catd);
}
```

Message Catalogs and the File System

There are no standard conventions for the location and naming of message catalogs; these are left to the application. In general, applications might choose either to locate message catalogs within a subtree corresponding to the supported language, /*application_name*/$LANG/*.cat, or to consolidate all message catalogs in one sub-directory, /*application_name*/catalogs/*.cat.

The environmental variable, NLSPATH allows this flexibility, Its use is as follows:

```
NLSPATH = /appl_lib/%L/%N.cat:/nlslib/%N/%L
```

A substitution field is introduced by %, with %L substituting for the current value of LANG, and %N, substituting for the value of the name parameter used in the call to catopen(). catopen() searches first in /appl_lib/$LANG/cat_name.cat, and then in /nlslib/cat_name/$LANG for the message catalog.

Generally, the use of NLSPATH is discouraged, as it leads to the users having uncertain knowledge of the location of the message catalog at run-time. It is preferred practice to use the default location for messages in /usr/share/lib/locale/LC_MESSAGES/*locale*/*name* In this case, collision of message catalogs should be determined by the application installation script.

Static and Dynamic Messaging

Assuming that the programmer uses the message retrieval facility as described in the previous section, it is still important to understand how best to define strings in the *original* form so that they can be easily translated at a later stage. The examples in this section do not contain references to catgets() (These are only removed for readability), however it is assumed that in the real case these calls would be surrounding the string literal itself.

Application writers can take two approaches to message creation, either *static* messaging or *dynamic* messaging. Static message usage involves pre-formatted messages which are selected from a message catalog and printed without reordering by the application. Dynamic message creation also selects messages from a message catalog, but orders and assemble messages at run-time instead of statically presenting them. 4.1 provides C language routines for both strategies.

The advantage of static messaging is its simplicity. A single message is selected from the catalog and is sent directly to the output stream. However, with static messaging, care must be taken to avoid splitting a message across *printf()* statements. Otherwise the message will be difficult to translate. This is illustrated below:

```
/* Poor practice: Do Not Split Messages   */
printf("This sentence may be difficult to translate ");
printf("because it spans multiple printf statements.\n");
```

Better practice is to place entire sentences within a single `printf()` statement, as shown below:

```
/* Good practice: 1 Message Per Sentence   */
printf("This sentence is easy to translate \
    because it is included with one printf statement.\n");
```

Another problem that can arise is when a `printf()` statement could result in more than one sentence when executed. The illustration below demonstrates a message that would not be translatable.

```
/* Poor practice:  Mixing Multiple Sentences   */
printf("%- Insufficient resources to%s %d%s resource%s - %s",
    func, (alloc_flg ? " allocate" : "reserve"),
    count, (request_flg ? " sufficient" : ""),
    (count == 1 ? "" : "s"), "Request failed.");
```

One solution is to split the message into separate print statements, one per variant of the message, and to have an implicit switch statement that selects the correct version at run-time.

Dynamic messaging can be used when the exact content or order of a message is not known until run-time. Unless done carefully, this approach can cause translation problems. If the positional dependence of keywords is hard-coded into the program, the program itself must also be changed for the message to be successfully translated. Obviously, this defeats the purpose of message catalogs.

The solution is a set of routines which enables proper dynamic message creation by allowing the calculation of string arguments to be performed in position-independent manner. The need for this will now be illustrated.

```
/* Poor practice:   Position Dependent Keywording   */
printf("Unable to %s the %s\n",
 (lock_flg?"lock":"find"), (type_flg?"page":"record"));
```

This program could alternatively execute in English as either:

```
Unable to lock that page.
Unable find that record.
```

However, the program's message could not be translated into the equivalent German,

```
"Das Programm kann die Seite nicht sperren."
```

and

```
"Das Programm kann der Rekord nicht finden."
```

because the German conventions for word order require that the program's keywords be reversed.

Release 4.1 solves this with functions which support dynamic message ordering: `printf()`, `fprintf()`, `sprintf()`, `scanf()`, `fscanf()`, and `sscanf()`.

These functions make the position of the argument independent of the underlying input string. Position within the string is declared by an extension to the conversion character `%`. The sequence

`%`***n***`$`

where *n* is a decimal digit, is substituted for the conversion character. Conversions are subsequently applied to the *n*th argument in the argument list, rather than to the next unused argument. In the example above, the format string would contain the new positional arguments:

```
printf("The program cannot %1$s %2$s\n",
 (lockflg?"lock":"find"), (type_flg?"page":"record"));
```

The English message catalog becomes:

```
"Unable %1$s %2$s"
"lock"
"find"
"the page"
"the record"
```

While the German message catalog becomes:

```
"Das Programm kann %2$s nicht %1$s"
"sperren"
"finder"
"das Seite"
"der Rekord"
```

The routines `nl_printf()`, `nl_fprintf()`, `nl_sprintf()`, `nl_scanf()`, `nl_fscanf()`, and `nl_sscanf()` are also provided for XPG2 compatibility, but since their functionality has been subsumed by the `printf()` family in XPG3, the use of the `nl_*` variants is not recommended.

Finally, remember to allow messages to have variable lengths. Applications should not make assumptions about the space required to express a message. Messages originally written in English will often expand in length when translated into foreign languages. However, applications should also plan for messages which become shorter under translation as well.

Messages using parameters should be carefully considered; it may be necessary to re-position the parameter within the message to allow for differences in translation.

Other Programming Considerations

Graphical Characters

Graphical characters such as `&` and `!` are subject to widely differing interpretations and should be avoided. However, the `%` percentage symbol is widely understood.

Using menu selections or making choices with cursor position is a useful technique for making application programs independent of the locale in which the application runs. Choosing items by typing the first letter works less well.

Printing

Manufacturers of printers have lagged the manufacturers of computer systems in the incorporation of standard codesets within their products. Application programs should beware of printer-specific codesets which may not translate directly from the ISO 8859/1 codeset used in SunOS. Applications expecting to encounter such printers should define structures which contain the printer specific codesets and specifically translate files to be printed.

SunOS minimizes these problems by providing 8-bit clean datapaths within `lpr` and by also using the ISO 8859/1 codeset within the TranScript unbundled software product which drives Sun LaserWriter printers.

Page Sizes

The dimensions of the standard paper stocks used around the world varies widely, as shown below. Internationalized applications should not make assumptions about the pagesizes available to them. Release 4.1 provides no support for tracking the page size to be written by an application; this is the responsibility of the application program itself.

Table 6-8 *Common International Page Sizes*

Paper Size Name	Measurements(Inches)
Letter	8.5" X 11"
Legal	8.5" X 14"
A4	8.34" X 11.78"
JIS B4	10.20" X 14.43"
JIS B5	7.23" X 10.20"

The standard paper trays distributed with the Sun LaserWriter and LaserWriter II printers support letter, legal, and A4 sized trays.

The best strategy for the application is to make no assumptions about the page sizes available, and to delay formatting to page size until print time. Tables of explicitly supported page sizes should be used; default choices might be used with an optional user interface for selecting from the supported page sizes.

Fonts

Font support can be broken into two categories: *printer* fonts and *screen* fonts. Printer font management is the responsibility of the application. SunOS provides no programmatic interface for this function. However, TranScript, available as an unbundled software product, provides filters which convert the common UNIX outputs into Postscript based files and fonts.

Default screen fonts for use with SunView 1, are in the directory `/usr/lib/fonts/fixedwidthfonts`.

Handling Multi-Byte Characters

There is an assumption underlying the relationship between the English language and its representation in a codeset--the assumption that all characters can be represented in one byte. ASCII makes the further restriction of assuming everything can fit into 7-bits of one byte. This may not apply to non-English languages, particularly the Asian languages. Asian languages typically need more than one byte (usually 2 or 3) to uniquely identify a character.

Applications intended for use outside of Europe may need to use another fundamental type for character representation, rather than a `char`. Other fundamental types defined in the proposed ANSI C Language standard and supported in Release 4.1 are wide (2-byte) characters of type `wchar_t`, and multi-byte characters. This data type is defined in `<stddef.h>`.

In practice, a multi-byte character can be defined to span any number of bytes, however, they are typically encoded within the system as wide characters.

The `mbtowc()` function maps from a multi-byte character to a wide character. It determines the number of bytes within the multi-byte character, and identifies and stores the code for the value of type `wchar_t` corresponding to the multi-byte code.

Other functions for manipulating wide and multi-byte characters and strings are:

`mblen()` returns the number of bytes within the multi-byte character.

`wctomb()`
: performs the corresponding backwards conversion from `wchar` to multi-byte.

`mbstowc()`
: converts a multi-byte string to an array of `wchar_t`.

`wcstombs()`
: converts a string of wide characters to an array of multi-byte characters.

7

System V Compatibility Features

This chapter is intended for users and programmers who want to learn about System V compatibility features in Release 4.1 of the SunOS operating system.

7.1. Introduction

Release 4.1 provides Sun Workstation® users and programmers with nearly complete System V compatibility. Sun's compatibility package allows programmers to write software that conforms to the Base Level of Release 3 of the System V Interface Definition (SVID). This release represents another phase of joint efforts by Sun and AT&T to unify versions of the UNIX operating system. The two principal versions have been BSD (now 4.3 BSD),† and System V in its various releases.

System V and 4.3 BSD are not radically different in architecture, the interface they present to the user, or the routines they provide for the programmer. Both are derived from the UNIX system originated by Ken Thompson and Dennis Ritchie in the mid-seventies; many features are essentially unchanged since then.

The System V compatibility package permits programmers to write and test software targeted for either System V Release 3, or 4.3 BSD. Commands, system calls, and library routines and headers can be drawn concurrently from either the Berkeley or the System V set. For users, it is even possible to have one window that uses System V by preference, and another window that uses BSD by preference (by placing `/usr/5bin` ahead of `/usr/ucb` in the shell's execution path, or vice versa).

Future Directions

Along with providing substantial conformance to the SVID Issue 2, the System V compatibility package in Release 4.1 also conforms to IEEE Standard 1001.3-1988 (POSIX.1). (See the chapter entitled *POSIX Conformance*, for details.)

4.1 also provides an additional compatibility package to conform with the *X/OPEN Portability Guide, Issue 2* (XPG2). Refer to the chapter entitled, *X/OPEN Compatibility Features* for more information about The packageX/OPENcompatibility conformance with the System V Verification Suite 3 (SVVS3). X/OPEN.

Further developments have brought SVID89, and Issue 3 of the X/OPEN Programmer's Guide (XPG3) closer together. These changes introduce

† An outgrowth of research at U.C. Berkeley, BSD stands for Berkeley Software Distribution.

differences between SVID Issue 2 and SVID89, as well as differences between XPG2 and XPG3.

These developments have complicated the compatibility situation overall. However, selecting the desired compatibility characteristics is simply a matter of properly constructing the shell's search path.

Commands in /usr/5bin provide the command-level interface required for System V Release 3, as defined by the SVID Issue 2. The System V C compiler, /usr/5bin/cc, links with libraries found in /usr/5lib, which substantially conform to SVVS3. For System V compatibility, place /usr/5bin ahead of /usr/bin in the shell's search path.

The cc command in /usr/xpg2bin uses the System V C compiler, and links with supplementary libraries found in /usr/xpg2lib. For X/OPEN compatibility and strictest conformance with SVVS3, place /usr/xpg2bin ahead of /usr/5bin.

For compatibility with SVID89 and POSIX, omit /usr/xpg2bin from the path. To sum up:

/usr/5bin:/usr/bin:/usr/ucb:...
: For conformance with System V Release 3, POSIX.1 and new functionality in SVID89 and XPG3.

/usr/xpgbin:/usr/5bin:/usr/bin:/usr/ucb:...
: For strict conformance with SVVS3, SVID Issue 2 and XPG2.

/usr/ucb:/usr/bin:...
: For traditional BSD compatibility.

System V Enhancements

Unless otherwise noted below, Release 4.1 incorporates the full functionality of the SVID Issue 2 Base Level system. The compatibility package features:

- A number of system calls that are compatible with SVID Base Level, including: chown(), creat(), fcntl(), kill(), mknod(), open(), and utime().
- The complete System V STREAMS interface, to support portable communication protocol modules, and to simplify the writing of device drivers.
- The TLI transport-level networking interface. (Refer to *UNKNOWN TITLE ABBREVIATION: TRANSPORT* for more information.)
- RFS remote file sharing. (Refer to *System and Network Administration* for more information.)
- A STREAMS-based tty(4) interface that is fully System V and BSD compatible, which supports all character sizes and parity settings. (For an introduction to STREAMS and STREAMS-related facilities, see the *STREAMS Programming* manual).
- A System V compatible version of the archive utility ar(1V).
- System V batch utilities and job scheduling facilities: at(1), batch(1), cron(1), and crontab(1).

- Access to Sun's value-added libraries (SunView for example) from inside System V programs.
- System V IPC facilities, including messages, semaphores, and shared memory segments. For more information about these facilities, refer to *Programming Utilities and Libraries.*
- System V first-in-first-out (FIFO) files, also called *named pipes*, which allow unrelated processes to communicate as if within a pipeline. (FIFO files are created using the `mknod()` system call.)
- The `lockf`(3) library routine for mandatory file and record locking.
- Password aging.
- A line printer command interface that is compatible with System V, and works with the system's BSD-based printer subsystem.
- SVID-compliant versions of memory-allocation routines, supplied in the `libmalloc.` library in `/usr/5lib`.
- SVID-compliant versions of math library routines, in the `svidm` library in `/usr/5lib`. The standard math library conforms to ANSI/IEEE Standard 754-1985.
- System V accounting. (Refer to *System and Network Administration* for more information.)

How the Compatibility Features Work

System V programs that are upwards compatible with those in 4.3 BSD have already been added to the regular system directories. For example, `/usr/bin/sh` is the new Bourne shell, and `/usr/bin/make` includes backward-compatible System V enhancements.

Programs that existed only on System V have been added to regular system directories as well. For example, the text manipulation programs `cut`(1) and `paste`(1) both reside in `/usr/bin`.

System V programs that are incompatible with those in 4.3 BSD reside in the directory `/usr/5bin`. For example, `/usr/5bin/stty` has an entirely different set of options from `/usr/bin/stty`. If you want to use System V programs by preference, simply include `/usr/5bin` early in your path, as in these lines from the `.login` or `.profile` files:

```
(csh) set path = (/usr/5bin /usr/bin /usr/ucb .)

(sh)  PATH=/usr/5bin:/usr/bin:/usr/ucb::
      export PATH
```

The directories `/usr/5bin`, `/usr/5lib`, and `/usr/5include` contain material that has not yet been converged. Libraries and include files for compiling System V software reside in `/usr/5lib` and `/usr/5include` respectively. These libraries and headers are not compatible with their counterparts in `/usr/include` or `/usr/lib`.

If you want to compile a program written for System V, don't use `/usr/bin/cc` but rather `/usr/5bin/cc`, which will read all the correct include files and load the correct libraries.

The directories that constitute the System V compatibility package are optional. The `suninstall`(8) program lets you decide whether or not to load these directories.

File-Creation Group ID Semantics

SunOS operating system releases prior to 4.0 used BSD group-ID assignment semantics for file creation. Under this scheme, a file is assigned the group ID (GID) of the directory in which it is created. By contrast, under System V a file is assigned the GID of the creating process. SunOS system Release 4.0 and later releases (including 4.1) allow users to select either of the two group-ID assignment schemes. When a directory has its set-GID bit set, the BSD semantics are in effect; a file created in that directory will be assigned the directory's GID. Otherwise, it will be assigned the effective GID of the creating process (System V semantics).

A newly created directory inherits the value of its parent's set-GID bit.

Release 4.1 distribution tapes are shipped with the set-GID bit set on all directories, thereby giving BSD semantics as the default. When you install Release 4.1, if you want to mount old filesystems and have them act as they did in the past, type the following command line for each mounted file system:

```
# find mounted.directory -type d -exec chmod g+s "{}" \;
```

To set System V semantics on some portion of the installed system, use `g-s` instead of `g+s` in the above command line. There is a mount option called `grpid` that always provides BSD semantics. This option may be needed when a client system running Release 4.0 or later mounts a file system from a server that has not yet been upgraded to a 4.x release.

Ancillary Libraries

In addition to the C library in `/usr/5lib` Release 4.1 supplies the following libraries for more complete compliance with the SVID:

- The `libmalloc` library contains versions of memory-allocation routines such as `malloc()`, that return the errors expected by the SVVS (System V Verification Suite). The default routines return different errors under certain conditions. To select the System V versions of these routines, compile your program with the `-llibmalloc` option to '`cc`'.
- The `svidm` library is a System V implementation of the math library. The default implementation conforms strictly to the IEEE Standard 754-1985 for floating-point arithmetic. To select the System V version, compile your program with '`-lsvidm`'.

7.2. SVID Compliance

The tables in this section illustrate how Release 4.1 complies with Issue 2 of the System V Interface Definition (SVID).

Table 7-1 *SVID Base System OS Service Routines*

SVID Base System OS Service Routines						
Non-Compliant	SVID-Compliant in 4.1					
	`_exit()`	`execlp()`	`free()`	`kill()`	`readdir()`	`unlink()`
	`abort()`	`execv()`	`freopen()`	`link()`	`realloc()`†	`ustat()`
`access()`	`alarm()`	`execve()`	`fseek()`	`lseek()`	`rewind()`	`utime()`
`chown()`	`calloc()`†	`execvp()`	`fstat()`	`mallinfo()`†	`setuid()`	`wait()`
`fcntl()`	`chdir()`	`exit()`	`ftell()`	`malloc()`†	`sigset()`	
`getcwd()`	`chmod()`	`fclose()`	`fwrite()`	`mallopt()`†	`sleep()`	
`lockf()`	`clearerr()`	`fdopen()`	`getegid()`	`mkdir()`	`stat()`	
`mount()`	`close()`	`feof()`	`geteuid()`	`mknod()`	`stime()`	
`read()`	`closedir()`	`ferror()`	`getgid()`	`open()`	`sync()`	
`rmdir()`	`creat()`	`fflush()`	`getpgrp()`	`opendir()`	`system()`	
`write()`	`dup()`	`fileno()`	`getpid()`	`pause()`	`time()`	
	`dup2()`	`fopen()`	`getppid()`	`pclose()`	`ulimit()`	
	`execl()`	`fork()`	`getuid()`	`pipe()`	`umask()`	
	`execle()`	`fread()`	`ioctl()`	`popen()`	`uname()`	
†When compiled with `-llibmalloc`.						

Table 7-2 *SVID Base System General Library Routines*

SVID Base System General Library Routines						
SVID-compliant in 4.1						
`_tolower()`	`erand48()`	`gmtime()`	`jrand48()`	`printf()`	`step()`	`tfind()`
`_toupper()`	`erf()`	`gsignal()`	`lcong48()`	`putc()`	`strcat()`	`tmpfile()`
`abs()`	`erfc()`†	`hdestroy()`	`ldexp()`	`putchar()`	`strchr()`	`tmpnam()`
`acos()`†	`exp()`†	`hsearch()`	`lfind()`	`putenv()`	`strcmp()`	`toascii()`
`advance()`	`fabs()`†	`hypot()`†	`localtim()`	`puts()`	`strcpy()`	`tolower()`
`asctime()`	`fgetc()`	`infinity()`†	`log()`†	`putw()`	`strcspn()`	`toupper()`
`asin()`†	`fgets()`	`isalnum()`	`log10()`†	`qsort()`	`strdup()`	`tsearch()`
`atan2()`†	`floor()`†	`isalpha()`	`longjmp()`	`rand()`	`strlen()`	`ttyname()`
`atof()`	`fmod()`	`isascii()`	`lrand48()`	`scanf()`	`strncat()`	`twalk()`
`atoi()`	`fprintf()`	`isatty()`	`lsearch()`	`seed48()`	`strncmp()`	`tzset()`
`atol()`	`fputc()`	`iscntrl()`	`matherr()`†	`setbuf()`	`strncpy()`	`ungetc()`
`bsearch()`	`fputs()`	`isdigit()`	`memccpy()`	`setjmp()`	`strpbrk()`	`vfprintf()`
`ceil()`†	`frexp()`	`isgraph()`	`memchr()`	`setkey()`	`strrchr()`	`vprintf()`
`clock()`	`fscanf()`	`islower()`	`memcmp()`	`setvbuf()`	`strspn()`	`vsprintf()`
`compile()`	`ftw()`	`isprint()`	`memcpy()`	`sin()`†	`strtod()`	`y0()`†
`cos()`†	`gamma()`†	`ispunct()`	`memset()`	`sinh()`†	`strtok()`	`y1()`†
`cosh()`†	`getc()`	`isspace()`	`mktemp()`	`sprintf()`	`strtol()`	`yn()`†
`crypt()`	`getchar()`	`isupper()`	`modf()`	`sqrt()`†	`swab()`	
`ctermid()`	`getenv()`	`isxdigit()`	`mrand48()`	`srand()`	`tan()`†	
`ctime()`	`getopt()`	`j0()`†	`nrand48()`	`srand48()`	`tanh()`†	
`drand48()`	`gets()`	`j1()`†	`perror()`	`sscanf()`	`tdelete()`	
`encrypt()`	`getw()`	`jn()`†	`pow()`†	`ssignal()`	`tempnam()`	
†When compiled with `-lsvidm`.						

Table 7-3 *SVID Kernel Extension OS Service Routines*

SVID Kernel Extension OS Service Routines			
SVID-compliant in 4.1			
acct()	msgsnd()	semctl()	
chroot()	nice()	semget()	shmdt()
msgctl()	plock()	semop()	shmget()
msgget()	profil()	shmat()	
msgrcv()	ptrace()	shmctl()	

Table 7-4 *SVID Basic Utilities Extension*

SVID Basic Utilities Extension						
Non-Compliant	SVID-compliant in 4.1					
	ar	comm	expr	mv	rmail	tee
	awk	cp	false	nl	rmdir	test
	banner	cpio	file	nohup	rsh	touch
	basename	cut	find	pack	sed	tr
ps	cal	date	grep	paste	sh	true
	calendar	df	kill	pcat	sleep	umask
	cat	diff	line	pg	sort	uname
	cd	dirname	ln	pr	spell	uniq
	chmod	du	ls	pwd	split	unpack
	cmp	echo	mail	red	sum	wait
	col	ed	mkdir	rm	tail	wc

Table 7-5 *SVID Advanced Utilities Extension*

SVID Advanced Utilities Extension					
Non-Compliant	SVID-compliant in 4.1				
	at	cu	logname	su	
	batch	dd	lp	tabs	uustat
mailx	cancel	dircmp	lpstat	tar	uuto
shl	chgrp	egrep	mesg	tty	uux
who	chown	ex	newgrp	uucp	vi
	cron	fgrep	od	uulog	wall
	crontab	id	passwd	uuname	write
	csplit	join	stty	uupick	

Table 7-6 *SVID Administered Systems Extension Utilities*

SVID Administered Systems Extension Utilities				
Non-Compliant		SVID-Compliant		
fsck	**sadc**	acctcms	clri	nice
fsdb	**sadp**	acctcom	devnm	prctmp
fuser	**sar**	acctcon1	diskusg	prdaily
init	**setmnt**	acctcon2	dodisk	prtacct
killall	**sysdef**	acctdisk	fwtmp	pwck
labelit	**timex**	acctmerg	grpck	runacct
link	**unlink**	accton	ipcrm	shutacct
mdfs	**volcopy**	acctprc1	ipcs	startup
mount	**whodo**	acctprc2	lastlogin	sync
mvdir		acctwtmp	mknod	turnacct
sa1		chargefee	monacct	umount
sa2		ckpacct	ncheck	wtmpfix

Table 7-7 *SVID Software Development Extension Utilities*

SVID Software Development Extension Utilities			
Non-Compliant	SVID-Compliant in 4.1		
as	admin	lex	tsort
dis	cc	lint	unget
ld	chroot	lorder	val
nm	cflow	m4	what
prof	cpp	make	xargs
sdb	cxref	prs	
size	delta	rmdel	
strip	env	sact	
yacc	get	time	

Table 7-8 *SVID Software Development Extension Additional Routines*

SVID Software Development Extension Additional Routines			
Non-Compliant	SVID-Compliant in 4.1		
endutent()	a64l()	getgrnam()	monitor()
getutent()	assert()	getlogin()	nlist()
getutid()	endgrent()	getpass()	putpwent()
getutline()	endpwent()	getpwent()	setgrent()
pututline()	fgetgrent()	getpwnam()	setpwent()
setutent()	fgetpwent()	getpwuid()	sgetl()
utmpname()	getgrent()	l64a()	sputl()
	getgrgid()	mark()	

Table 7-9 *SVID Terminal Interface Extension Utilities*

SVID Terminal Interface Extension Utilities	
SVID-compliant in 4.1	
tic	put

Table 7-10 *SVID Terminal Interface Extension Library Routines*

SVID Terminal Interface Extension Library Routines				
SVID-compliant in 4.1				
addch()	getstr()	mvwgetstr()	scr_dump()	vidattr()
addstr()	gettmode()	mvwin()	scr_init()	vidputs()
attroff()	getyx()	mvwinch()	scr_restore()	waddch()
attron()	halfdelay()	mvwinsch()	scroll()	waddstr()
attrset()	has_ic()	mvwprintw()	scrollok()	wattroff()
baudrate()	has_il()	mvwscanw()	set_term()	wattron()
beep()	idlok()	napms()	setscrreg()	wattrset()
box()	inch()	newpad()	setterm()	wclear()
cbreak()	initscr()	newterm()	setupterm()	wclrtobot()
clear()	insch()	newwin()	slk_clear()	wclrtoeol()
clearok()	insertln()	nl()	slk_init()	wdelch()
clrtobot()	intrflush()	nocbreak()	slk_label()	wdeleteln()
clrtoeol()	keyname()	nodelay()	slk_noutrefresh()	wechochar()
copywin()	keypad()	noecho()	slk_refresh()	werase()
def_prog_mode()	killchar()	nonl()	slk_restore()	wgetch()
def_shell_mode()	leaveok()	noraw()	slk_set()	wgetstr()
delay_output()	longname()	overlay()	slk_touch()	winch()
delch()	move()	overwrite()	standend()	winsch()
deleteln()	mvaddch()	pechochar()	standout()	winsertln()
delwin()	mvaddstr()	pnoutrefresh()	subpad()	wmove()
doupdate()	mvcur()	prefresh()	subwin()	wnoutrefresh()
echo()	mvdelch()	printw()	tgetent()	wprintw()
echochar()	mvgetch()	putp()	tgetflag()	wrefresh()
endwin()	mvgetstr()	raw()	tgetnum()	wscanw()
erase()	mvinch()	refresh()	tgetstr()	wsetscrreg()
erasechar()	mvinsch()	reset_prog_mode()	tgoto()	wstandend()
fixterm()	mvprintw()	reset_shell_mode()	touchline()	wstandout()
flash()	mvscanw()	resetterm()	touchwin()	
flushinp()	mvwaddch()	resetty()	tparm()	
getbegyx()	mvwaddstr()	saveterm()	tputs()	
getch()	mvwdelch()	savetty()	typeahead()	
getmaxyx()	mvwgetch()	scanw()	unctrl()	

Table 7-11 *SVID Open Systems Networking Interfaces (TLI) Library Routines*

SVID Open Systems Networking Interfaces (TLI) Library Routines			
SVID-compliant in 4.1			
t_accept()	t_getinfo()	t_rcvdis()	t_sndrel()
t_alloc()	t_getstate()	t_rcvrel()	t_sndudata()
t_bind()	t_listen()	t_rcvudata()	t_sync()
t_close()	t_look()	t_rcvuderr()	t_unbind()
t_connect()	t_open()	t_revconnect()	
t_error()	t_optmgmt()	t_snd()	
t_free()	t_rcv()	t_snddis()	

Table 7-12 *SVID STREAMS I/O Interface Operating System Service Routines*

SVID STREAMS I/O Interface Routines		
SVID-compliant in 4.1		
getmsg()	poll()	putmsg()

Table 7-13 *SVID Shared Resource Environment (RFS) Utilities*

SVID Shared Resource Environment (RFS) Utilities			
SVID-compliant in 4.1			
adv	fusage	rfadmin	rfstop
dname	idload	rfpasswd	rmnstat
fumount	nsquery	rfstart	unadv

8

X/OPEN Compatibility Features

This chapter describes the X/OPEN compatibility features in Release 4.1 of the SunOS operating system.

8.1. Introduction

The X/OPEN compatibility package allows programmers to write software that conforms to the base level of the X/OPEN 1987 standard. The System V versions of most required commands, system calls, library routines, and headers conform to the *X/OPEN Programmer's Guide* (1987) definition (XPG-2), For routines and headers that do not, Release 4.1 provides X/OPEN conforming versions in `/usr/xpg2lib`, and `/usr/xpg2include`.

To compile C programs that conform to the X/OPEN standard, you can use the `cc` executive script in `/usr/xpg2bin`. To use this as the preferred compiler, place `/usr/xpg2bin` ahead of `/usr/5bin` and `/usr/bin` in the shells execution path. (See also *System V Compatibility Features*, in this manual, for more information about System V.)

Ancillary Libraries

In addition to the System V and XPG libraries, Release 4.1 supplies the following ancillary libraries for compliance with the SVID:

- The `libmalloc` library (in `/usr/5lib`) contains versions of memory-allocation routines such as `malloc()`, that return the errors expected by the SVVS (System V Verification Suite). The default 4.1 routines return different errors under certain conditions. To select the System V versions of these routines, compile your program with the `-llibmalloc` option to '`cc`'.
- The `svidm` library is a System V implementation of the math library. The default 4.1 implementation conforms strictly to the IEEE Standard 754-1985 for floating-point arithmetic. To select the System V version, compile your program with '`-lsvidm`'.

8.2. X/OPEN Conformance

The tables in this section illustrate how Release 4.1 conforms to X/OPEN (1987). These tables account for all commands, routines and files described in Volumes 1 and 2 of XPG-2.

Figure 8-1 *System Calls*

Optional	Required					
Non-Conforming	Conforming					
acct(2)	access(2)	execl(2)	fork(2)	mount(2)	stat(2)	uname(2)
brk(2)	alarm(2)	execle(2)	getpid(2)	open(2V)	stime(2)	unlink(2)
chroot(2)	chdir(2)	execlp(2)	getuid(2)	pause(2)	sync(2)	ustat(2)
nice(2)	chmod(2)	execv(2)	ioctl(2)	pipe(2)	time(2)	utime(2)
plock(2)	chown(2)	execve(2)	kill(2V)	read(2V)	times(2)	wait(2)
profil(2)	close(2)	execvp(2)	link(2)	setpgrp(2V)	ulimit(2)	write(2V)
ptrace(2)	creat(2)	exit(2)	lseek(2)	setuid(2)	umask(2)	
	dup(2)	fcntl(2V)	mknod(2)	signal(2)	umount(2)	

Figure 8-2 *Subroutines and Libraries*

Non-Conforming	Conforming				
Optional		NLS	Required		
			abs(3)	getlogin(3)	qsort(3)
			assert(3)	getopt(3)	rand(3V)
			bsearch(3)	getpass(3)	regexp(3)
			clock(3C)	getpw(3)	setbuf(3S)
	bessel(3M)		crypt(3)	getpwent(3)	setjmp(3)
	end(3C)[1]	conv(3V)[3]	ctermid(3S)	gets(3S)	ssignal(3)
	erf(3M)	ctime(3)	cuserid(3S)	getut(3C)	stdio(3S)[6]
	exp(3M)	ctype(3V)	directory(3)	hsearch(3)	strtol(3)
gamma(3M)	floor(3M)[2]	ecvt(3)[4]	drand48(3)	l3tol(3C)	swab(3)
	hypot(3M)	printf(3S)	econvert(3)	lockf(3)	system(3)
	matherr(3M)	scanf(3S)	fclose(3S)	logname(3)	tmpfile(3S)
	monitor(3)	string(3)	ferror(3S)	lsearch(3)	tmpnam(3S)
	sinh(3M)	strtod(3)	fopen(3S)	malloc(3)[5]	tsearch(3)
	trig(3M)		fread(3S)	memory(3)	ttyname(3)
			frexp(3M)	mktemp(3)	ttyslot(3)
			fseek(3S)	perror(3)	ungetc(3S)
			ftw(3)	popen(3S)	vprintf(3S)
			getc(3S)	putc(3S)	
			getcwd(3)	putenv(3)	
			getenv(3)	putpwent(3)	

[1]Data items, not routines.
[2]Routines documented in rint(3M).
[3]Routines documented in ctype(3S).
[4]Routines documented in econvert(3).
[5]When compiled with -llibmalloc.
[6]Overview of library, not routines.

Figure 8-3 *File Formats*

Non-Conforming	Conforming
acct(5)	cpio(5)
utmp(5)	group(5)
	passwd(5)

Figure 8-4 *Headers*

Conforming
<sys/acct.h>
<assert.h>[1]
<ctype.h>[1]
<sys/dirent.h>[2]
environ(5)[3]
<errno.h>
<fcntl.h>[1]
<ftw.h>
<grp.h>
<limits.h>[2]
<sys/lock.h>
<malloc.h>[1]
<math.h>[2]
<memory.h>
<mon.h>
<pwd.h>
<search.h>
<setjmp.h>
<signal.h>
<sys/stat.h>
<stdio.h>[2]
<string.h>
<termio.h>
<time.h>[1]
<sys/times.h>
<sys/types.h>
<unistd.h>
<ustat.h>
<utmp.h>[2]
[1]Filename relative to /usr/5include. [2]Filename relative to /usr/xpg2include. [3]Global data format, not a header.

Figure 8-5 *Commands*

Optional	Required						
Non-Conforming		Conforming					
		ar	dd	join	od	su	uux
		at	delta	kill	pack	sum	val
		awk	df	ld	passwd	tabs	vi
		banner	diff	lex	paste	tail	wait
		basename	dircmp	line	pg	tar	wall
as	**batch**	cat	dirname	lint	pr	tee	wc
dis	**cal**	chown	du	logname	prs	test	what
mailx	**calendar**	chroot	echo	lorder	pwd	time	write
mknod	**cancel**	cmp	ed	lp	rm	touch	xargs
newgrp	**cc**	col	egrep	lpstat	rmdel	tr	yacc
news	**cd**	comm	env	ls	sact	true	
prof	**cflow**	cp	ex	m4	sed	tsort	
sdb	**chgrp**	cpio	expr	mail	sh	tty	
shl	**chmod**	cpp	false	make	size	umask	
who	**ps**	crontab	fgrep	mesg	sleep	uname	
		csplit	file	mkdir	sort	unget	
		cu	find	mv	spell	uniq	
		cut	get	nl	split	uucp	
		cxref	grep	nm	strip	uustat	
		date	id	nohup	stty	uuto	

Figure 8-6 *Special Files*

Optional	Required
Non-Conforming	Conforming
sct(7†)	console(4S) null(4) termio(4) tty(4)
†Section 7 of XPG-2, Volume 2.	

9

POSIX Conformance

Conformance with IEEE Standard 1003.1-1988

Release 4.1 of the SunOS operating system is a conforming implementation as defined in Section 2.2.1.1 (*Requirements*) of the *Portable Operating System Interface for Computer Environments* (POSIX), IEEE Standard 1003.1 (POSIX.1).

Scope

To comply with Section 2.2.2.1 (*Documentation*), this chapter describes the behavior of features in Release 4.1 which are described in the POSIX.1 standard as implementation-defined, or for which it is stated that implementations may vary. It does not describe any extensions or enhancements outside the scope of the standard.

As required, this chapter also describes the contents of the `<limits.h>` and `<unistd.h>` headers, along with the conditions under which values defined in those files may vary, and the limits by which they may. (See *Headers*, below.)

This chapter follows the structure of the POSIX.1 standard, and is intended as a supplement to that document. For more detailed information about the behavior of the features mentioned herein, refer to the *SunOS Reference Manual*.

Implementation-Defined Features

POSIX.1 Section 2, Definitions and General Requirements

2.3 General Terms

For Release 4.1, with regard to the definition for *clock tick*, the constant `{CLK_TCK}` is defined to be 60 (intervals per second). (This is unlikely to change.)

2.4 General Concepts

In addition to the standard file status inquiries (refer to `stat`(2) in the *SunOS Reference Manual*), Release 4.1 provides the following macros:

`S_ISLNK()`	test for a symbolic link
`S_ISSOCK()`	test for a socket.

The values for `{NAME_MAX}` and `{PATH_MAX}` are retrieved using the `pathconf`(2) system call.

The constant `LS_ISVTXT` refers to the sticky bit. For a directory, this bit determines whether or not an unprivileged user may delete or rename another user's files (refer to chmod(2)).

2.5 Error Numbers

Routines generally return the error code of the first error they encounter. The operating system supports a number of error codes in addition to those defined in POSIX.1. Refer to intro(2) for the complete list of error codes in Release 4.1.

2.6 Primitive System Data Types

In addition to the standard global data types, 4.1 defines the following:

caddr_t	type to hold machine addresses
clock_t	clock ticks (units = 60ths of a second)
daddr_t	disk address type
key_t	used for System V IPC system calls
sigset_t	signal mask
speed_t	tty baudrates
tcflag_t	line discipline modes
time_t	time (units = seconds)
wchar_t	for wide characters (multi-byte)

2.7 Environment Description

Values for {ARG_MAX}, {NAME_MAX}, and other implementation-defined values described in this section are retrieved using the sysconf(2) or pathconf(2) system calls. (See the discussion of Section 3.1.2.2 for information about how {ARG_MAX} is obtained.) Initial values are set either to the minimum value specified in the standard, or are undefined.

2.10.3 Compile-Time Symbolic Constants for Portability Specifications

Both {_POSIX_JOB_CONTROL} and {_POSIX_SAVED_IDS} are defined to be 1 (that is, they are in effect) in 4.1.

POSIX.1 Section 3, Process Primitives

3.1 Process Creation and Execution

3.1.1 Process Creation

3.1.1.2

Any relevant characteristics not defined in the standard are inherited by the child process.

3.1.2 Execute a File

3.1.2.2 Description (execl(), execv(), execle(), execve(), execlp(), execvp())

{ARG_MAX} is retrieved using the sysconf() function. This value includes the total of bytes available for a new process's arguments, environment and stack. {ARG_MAX} also includes initial pointers into the argument and environment vectors.

The space required for the arguments, environment and stack by an execve() call is determined by the following formula:

$$space = (((na + 4) * bpw) + nc) + click$$

space is then rounded up to the next click boundary. A click is the number of bytes that the system's memory-management facilities treat as a single unit. On

Sun-4, Sun-3 and Sun386i systems, one click equals 8192 bytes. On Sun-2 systems, one click equals 2048 bytes.

na is the count of arguments and environment variables. *bpw* is the number of bytes per word: 4 on all Sun systems. *nc* is the count of bytes in the argument and the environment vectors, including null terminators, and rounded up to the next word boundary.

When `PATH` is not set, Release 4.1 supplies a default search path:

```
.:/usr/ucb:/bin:/usr/bin
```

3.1.2.4 Errors (`EACCESS`)

Release 4.1 supports executables in `a.out`(5) format. When the first line of a text file takes the form:

```
#! interpreter
```

the system invokes the named *interpreter* to interpret the file. As a special case, if the first character of the file is a pound-sign (#), Release 4.1 invokes the C shell (`/usr/bin/csh`). Otherwise, the system invokes a Bourne shell (`/usr/bin/sh`).

3.2 Process Termination

3.2.1 Wait for Process Termination

3.2.1.2 Description (`wait()`, `waitpid()`)

Release 4.1 assigns process ID 1 (the PID of the `init` process) as the new parent process ID (PPID) for an orphaned process.

Release 4.1 supports job control.

3.2.2 Terminate a Process

3.2.2.2 Description (`_exit()`)

Release 4.1 supports the `SIGCHLD` signal.

Release 4.1 assigns process ID 1 (the PID of the `init` process) as the new (PPID) for an orphaned process.

Release 4.1 supports job control.

3.3 Signals

3.3.1 Signal Concepts

3.3.1.1 Signal Names

Release 4.1 supports job control and the signals required for it. In addition to the signals defined in the standard, 4.1 supports the following:

`SIGBUS`	bus error
`SIGCLD`	System V name for `SIGCHLD`
`SIGEMT`	EMT instruction
`SIGIO`	asynchronous I/O available
`SIGIOT`	IOT instruction
`SIGLOST`	resource lost (such as a record-lock)
`SIGPOLL`	System V name for `SIGIO`
`SIGPROF`	profiling time alarm

SIGSYS	bad argument to system call (when kernel is compiled with -DCOMPAT)
SIGTRAP	trace trap (not reset when caught)
SIGURG	urgent condition on I/O channel (socket)
SIGVTALRM	virtual time alarm
SIGWINCH	window changed size
SIGXCPU	exceeded CPU time limit
SIGXFSZ	exceeded file size limit

3.3.1.2 Signal Generation and Delivery

In addition to the signals defined in the standard, Release 4.1 delivers the following signals when the indicated event occur.

SIGBUS	on parity or other hardware error
SIGCLD	when the status of a child process changes
SIGEMT	when a software trap instruction occurs
SIGIO	on asynchronous I/O
SIGIOT	used only with RFS
SIGLOST	when a lock is broken (see lockd(8))
SIGPOLL	on asynchronous I/O
SIGPROF	when a profiling alarm occurs
SIGSYS	used only when kernel is compiled with -DCOMPAT
SIGTRAP	used for tracing (see ptrace(2))
SIGURG	when out-of-band data arrives on a socket
SIGVTALRM	when a virtual time clock alarm occurs
SIGWINCH	when a window changes size
SIGXCPU	when a process exceeds its CPU time limit (software)
SIGXFSZ	when a write exceeds the file size limit (software)

3.3.2 Send a Signal to a Process (kill())

When the pid argument is specified as '(pid_t)-1', the signal is broadcast to all processes. The usual permission checks for signals still apply.

3.3.2.2 Description

{_POSIX_SAVED_IDS} is defined to be true. Release 4.1 allows a process to receive a signal if its effective user ID (EUID) or saved user ID is the same as the sending process's real user ID (UID) or EUID.

3.3.4 Examine and Change Signal Action

3.3.4.2 Description (sigaction())

Release 4.1 supports the SA_NOCLDSTOP flag, which when set, suppresses generation of SIGCHLD signals.

3.3.6 Examine Pending Signals

3.3.6.4 Errors (sigpending())

EFAULT	The address passed as an argument is not within the process's address space.

POSIX.1 Section 4, Process Environment

4.2 User Identification
4.2.2 Set User and Group IDs
4.2.2.2 Description (`setuid()`, `setgid()`)

`{_POSIX_SAVED_IDS}` is defined for Release 4.1.

4.2.4 Get User Name
4.2.4.2 Description (`getlogin()`, `cuserid()`)

The constant `L_cuserid`, is defined in `<stdio.h>` to be 9. This is the minimum number of bytes in the array pointed to by the argument to `cuserid()`.

4.2.4.3 Returns

If the argument to `cuserid()` is NULL, the value returned points to static data also used by `getpwnam()`. Subsequent calls to either routine may overwrite this data.

4.3 Process Groups
4.3.3 Set Process Group ID for Job Control
4.3.3.2 Description (`setpgid()`)

Release 4.1 supports job control.

4.4 System Identification
4.4.1 System Name
4.4.1.2 Description (`uname()`)

Each element of the `uname` structure is a 9-character array. To satisfy other requirements for longer nodenames, the `nodename[]` array is immediately followed by a `nodeext[]` array of length 56. System administrators must configure the system with nodenames no longer than 9 characters, including the trailing NULL character, to conform to POSIX.1. ' 'nodename[] is followed by '`nodeext[77]`'.

4.4.1.4 Errors

`EFAULT` The address passed as an argument is not within the process's address space.

4.5 Time
4.5.2 Process Times
4.5.2.2 Description (`times()`)

`{CLK_TCK}` is defined to be 60 (per second).

4.6 Environment Variables
4.6.1 Environment Access
4.6.1.2 Description (`getenv()`)

Under Release 4.1, `getenv()` points directly into the static `environ` variable. By writing in this variable, you may alter the process's environment, but not that of its parent.

4.7 Terminal Identification

4.7.1 Generate Terminal Pathname

4.7.1.3 Returns (`ctermid()`)

If the argument to `ctermid()` is NULL, the routine returns a pointer to an array which is static, and which may be overwritten by a subsequent call.

4.7.2 Determine Terminal Device Name

4.7.2.2 Description (`ttyname()`, `isatty()`)

`ttyname()` returns a pointer to an array which is static, and which may be overwritten by a subsequent call.

4.7.2.4 Errors

`EBADF` `fd` is not a valid open file descriptor.

`EIO` An I/O error occurred while reading from or writing to the file system.

POSIX.1 Section 5, Files and Directories

5.1 Directories

5.1.1 Format of Directory Entries (<`dirent.h`>)

The file system-independent format for directory entries in Release 4.1 is:

```
struct  dirent {
    off_t          d_off;          /* offset of next disk dir entry */
    unsigned long  d_fileno;       /* file number of entry */
    unsigned short    d_reclen;    /* length of this record */
    unsigned short    d_namlen;    /* length of string in d_name */
    char      d_name[255+1];       /* name (up to MAXNAMLEN + 1) */
};
```

5.1.2 Directory Operations

In Release 4.1, the `DIR` data type is implemented using a file descriptor, with the attendant restrictions.

5.2 Working Directory

5.2.1 Change Working Directory

5.2.1.4 Errors (`chidr()`)

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.3 General File Creation

5.3.1 Open a File

5.3.1.2 Description (`open()`)

`O_CREAT`
Bits other than `07777` (`0xfff`) are cleared from the file permission bits passed to `open()`.

In addition to the flags defined by the standard, we supply:

`O_NDELAY`
4.3 BSD or SVID Issue 2 no-delay semantics.

`O_SYNC`
Each write is synchronous; no write returns until data has been flushed to disk.

5.3.1.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.3.3 Set File Creation Mask

5.3.3.2 Description (`umask()`)

In addition to the file-permission bits, 4.1 allows the `setuid` (`S_ISUID`), `setgid`, (`S_ISGID`), and `sticky` (`S_ISVTX`) bits to be masked using the `cmask` argument. (See also, `chmod`(2).)

5.3.4 Link to a File

5.3.4.2 Description (`link()`)

Release 4.1 does not support hard links across file systems. Hard links to directories may be created only by processes with UID zero, that is by `root` (the super-user).

5.3.4.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.4 Special File Creation

5.4.1 Make a Directory

5.4.1.2 Description (`mkdir()`)

`mkdir()` ignores non-permission bits in the process's file-creation mask.

5.4.1.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.4.2 Make a FIFO Special File

5.4.2.2 Description (`mkfifo()`)

`mkfifo()` ignores non-permission bits in the process's file-creation mask.

5.5 File Removal

5.5.1 Remove Directory Entries

5.5.1.2 Description (`unlink()`)

Only `root` (the super-user) may use `unlink()` to remove a directory. User processes can use `rmdir()` to remove (empty) directories.

5.5.1.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.5.2 Remove a Directory

5.5.2.2 Description (`rmdir()`)

In Release 4.1, `rmdir()` returns an error if an attempt is made to remove the mount point of any mounted file system. Otherwise, any user process may remove its current working directory.

5.5.2.4 Errors

`EBUSY` The directory to be removed is the mount point for a mounted file system, or is being used by another process.

`ENAMETOOLONG`
See also the remarks on Section 2.7 in this chapter.

5.5.3 Rename a File

5.5.3.4 Errors (`rename()`)

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.6 File Characteristics

5.6.1 File Characteristics

In addition to the required fields, the 4.1 `stat` structure includes the following:

`dev_t`	`st_rdev;`	for block and character-special files
`int`	`st_spare1;`	expansion for atime
`int`	`st_spare2;`	expansion for mtime
`int`	`st_spare3;`	expansion for ctime
`long`	`st_blksize;`	i/o block size
`long`	`st_blocks;`	blocks used
`long`	`st_spare4[2];`	expansion

5.6.2 Get File Status

5.6.2.4 Errors (`stat()`, `fstat()`)

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.6.3 File Access

5.6.3.2 Description (`access()`)

`root` (the super-user) is granted all permissions except writing to a read-only file system.

5.6.3.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.6.4 Change File Modes

5.6.4.2 Description (`chmod()`)

If you are not a member of the file's group, the SGID bit is cleared, unless the EUID of the process is zero (the super-user).

Access permissions for open file descriptors that refer to files on local (UFS) or RFS-mounted file systems are not affected by `chmod()`. Access permissions for descriptors referring to files on NFS-mounted file systems may change as a result of a successful `chmod()` call.

5.6.4.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.6.5 Change Owner and Group of a File

5.6.5.2 Description (`chown()`)

`{_POSIX_CHOWN_RESTRICTED}` is true for native (UFS) file systems. Remote file systems may have different attributes, which can be determined using the `pathconf()` system call.

If the effective UID of the process is zero (that of `root`, the super-user), `chown()` does not alter the file's SUID and SGID bits. Otherwise, `chown()` clears these bits.

5.6.5.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

Release 4.1 supports only the variables described in the standard.

5.7.1.4 Errors

`ENAMETOOLONG`
See the remarks on Section 2.7 in this chapter.

POSIX.1 Section 6, Input and Output Primitives

6.4 Input and Output

6.4.1 Read from a File

6.4.1.2 Description (`read()`)

When `nbyte` is greater than `{INT_MAX}`, `read()` returns an error and transfers no data. When `nbyte` is zero, `read()` returns zero, and no data is transferred.

6.4.1.4 Errors

`EINTR` A read from a slow device was interrupted by the delivery of a signal before any data arrived.

`EIO` An I/O error occurred while reading from or writing to the file system, or the calling process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal, or the process is orphaned.

6.4.2 Write to a File

6.4.2.2 Description (`write()`)

When `nbyte` is greater than `{INT_MAX}`, `write()` returns an error and transfers no data. When `nbyte` is zero, `write()` returns zero, and transfers no data.

Write requests to a pipe of greater than `{PIPE_BUF}` may be interleaved with write requests of other processes in the case of a named pipe.

6.4.2.4 Errors

`EINTR` A write to a slow device was by the delivery of a signal interrupted before any was transferred.

`EIO` An I/O error occurred while reading from or writing to the file system, or the calling process is in a background process group and is attempting to write to its controlling terminal, and either the process is ignoring or blocking the SIGTTOU signal, or the process is orphaned.

6.5 Control Operations on Files

6.5.2 File Control (<sys/fcntl.h>)

In addition to the values defined in the standard, 4.1 supports the following flags for use with fcntl(2):

F_GETOWN	Get the PID or GPID of processes receiving SIGIO and SIGURG signals.
F_SETOWN	Get the PID or GPID of processes receiving SIGIO and SIGURG signals.
F_RGETLK	Test a remote lock to see if it is blocked.
F_RSETLK	Set or clear a remote lock.
F_CNVT	Convert a file handle to an open descriptor.
F_RSETLKW	Set or clear a remote lock (blocking).
F_UNLKSYS	Remove remote locks for a given system.
O_SYNC	Perform writes to disk immediately.
O_NDELAY	Nonblocking I/O, System V style.

6.5.2.2 Description (fcntl())

Advisory record locking operations for non-regular files are not supported in Release 4.1.

6.5.3 Reposition Read/Write File Offset

6.5.3.4 Errors (lseek())

EISPIPE The file descriptor is associated with a pipe, FIFO, or socket.

POSIX.1 Section 7, Device- and Class-Specific Functions

7.1 General Terminal Interface

7.1.1 Interface Characteristics

7.1.1.2 Process Groups

{_POSIX_JOB_CONTROL} is defined to be true.

7.1.1.3 The Controlling Terminal

When a session leader has no controlling terminal, opens a terminal that is not the controlling terminal of another session, and did not the specify O_NOCTTY flag to open(), that terminal becomes the controlling terminal for the session.

When a session leader has no controlling terminal, and issues an

```
ioctl(fd, TIOCSCTTY, 0)
```

call on a terminal that is not already a controlling terminal, that terminal becomes the controlling terminal for the session.

7.1.1.4 Terminal Access Control

Release 4.1 supports job control, and the SIGTTIN signal behaves as described in the standard.

7.1.1.5 Input Processing and Reading Data

The system limit for {MAX_INPUT} is defined to be the high-water mark of the first module the queues in the 4.1 STREAMS terminal environment.

The tty STREAMS module also supports flow control. However, if the sending process ignores these signals, it is possible for data to be lost.

7.1.1.6 Canonical Mode Input Processing

For local terminals, when {MAX_CANON} is exceeded, local terminals issue a BEL character and drops the extra characters.

7.1.1.8 Writing Data and Output Processing

Data is buffered for output by the tty STREAMS module.

7.1.1.9 Special Characters

Under 4.1, all terminal-control characters can be changed.

{_POSIX_VDISABLE} is set to 0.

7.1.2 Settable Parameters (<sys/termios.h>)

In addition to the members listed, the termios structure includes the field:

```
char c_line
```

7.1.2.2 Input Modes

In addition to the input mode masks listed in the standard, 4.1 supports the following:

IUCLC	Translate upper case input characters to lower case.
IXANY	Any character acts as the start character.
IMAXBEL	Ring bell when {MAX_CANNON} is exceeded.

The initial setting of the input mode flag is (the bitwise OR of):

```
BRKINT | ICRNL | IXON | ISTRIP
```

7.1.2.3 Output Modes

In addition to OPOST, Release 4.1 supports the following output control mode masks:

OLCUC	Map lower case to upper on output.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select new-line delays:
NL0	0
NL1	0000400
CRDLY	Select carriage-return delays:
CR0	0
CR1	0001000
CR2	0002000
CR3	0003000
TABDLY	Select horizontal-tab delays or expansion:
TAB0	0
TAB1	0004000
TAB2	0010000

TAB3	XTABS
XTABS	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	0
BS1	0020000
VTDLY	Select vertical-tab delays:
VT0	0
VT1	0040000
FFDLY	Select form-feed delays:
FF0	0
FF1	0100000
PAGEOUT	(unimplemented)
WRAP	(unimplemented)

The initial setting for the output control flag `oflag` is:

```
OPOST | ONLCR | XTABS
```

7.1.2.4 Control Modes

In addition to the control mode masks listed in the standard, 4.1 supports the following:

CBAUD	Baud rate
LOBLK	*unimplemented*
CIBAUD	Input baud rate.
CRTSCTS	Enable RTS/CTS flow control.

7.1.2.5 Local Modes

In addition to the local mode masks listed in the standard, 4.1 supports the following:

XCASE	Canonical upper/lower presentation.
ECHOCTL	Echo control characters as '`^`*C*', delete character as '`^?`'.
ECHOPRT	Echo erase character as character erased.
ECHOKE	BS-SP-BS erase entire line on line kill.
DEFECHO	(unimplemented)
FLUSHO	Output is being flushed.
PENDIN	Retype pending input at next read or input character.

The initial setting for the local mode flag `lflag` is:

```
ISIG | ICANON | ECHO | IEXTEN
```

7.1.2.6 Special Control Characters

In addition to the control characters listed in the standard, 4.1 supports the following:

SWITCH	Switch shell layers character.
DSUSP	Delayed suspend (not supported).
REPRINT	Reprint the command line.
DISCARD	Temporarily discards output.
WERASE	Word erase.
LNEXT	Literal next, that is, quote the next character.

STATUS	Status (unimplemented).

The initial values for control characters in Release 4.1 are:

INTR	Control-C
QUIT	Control-\
ERASE	Control-?
KILL	Control-U
EOF	Control-D
EOF2	*no default character*
SWITCH	*not supported*
START	Control-Q
STOP	Control-S
SUSP	Control-Z
DSUSP	Control-Y
REPRINT	Control-R
DISCARD	Control-O
WERASE	Control-W
LNEXT	Control-V
STATUS	*not supported*

7.2 General Terminal Interface Control Functions

Job control is supported in Release 4.1.

7.2.2 Line Control Functions

7.2.2.2 Description (`tcsendbreak()`, `tcdrain()`, `tcflush()`, `tcflow()`)

On non-asynchronous transmissions, `tcsendbreak()` does not sent a break; it simply returns. For a delay of $n > 0$, `tsendbreak()` behaves as if it had been called n times.

7.2.3 Get Foreground Process Group ID

7.2.3.2 Description (`tcgetpgrp()`)

{`_POSIX_JOB_CONTROL`} is defined for Release 4.1, and `tcgetpgrp()` functions as described in the standard.

7.2.4 Set Foreground Process Group ID

7.2.4.2 Description (`tcgetpgrp()`)

{`_POSIX_JOB_CONTROL`} is defined for Release 4.1, and `tcsetpgrp()` functions as described in the standard.

POSIX.1 Section 8, Language-Specific Services for the C Programming Language

8.1 Referenced C Language Routines

8.1.1 Extensions to Time Functions

Release 4.1 ignores the :*value* format of the TZ environment variable.

8.1.2 Extensions to setlocale()

8.1.2.2 Description

In addition to the categories (environment variables) described in the standard, 4.1 supports the following:

LC_MESSAGES	Allows for display of alternate message texts.
LC_default	Allows for a default language other than the "C" environment when LANG is not set or is empty.

8.2 FILE-Type C Language Functions

8.2.2 Open a Stream on a File Descriptor

8.2.2.4 Errors (fdopen())

EINVAL The file descriptor is less than zero or greater than or equal to {OPEN_MAX}.

The type argument does not begin with 'a', 'r', or 'w'.

ENOMEM The function could not allocate memory for the required stream pointer.

POSIX.1 Section 9, System Databases

9.1 System Databases

The system default for the initial working directory is '/'. The default for the shell is /usr/bin/sh.

There is an additional password and comments field in the passwd database. There is an additional password field in the group database.

9.2 Database Access

9.2.1 Group Database Access

9.2.1.4 Errors (getgrgid(), getgrnam())

These routines depend on malloc(3) and fopen(2), either of which may fail and return an error.

9.2.2 User Database Access Functions

9.2.2.2 Description (getpwnam(), getpwuid())

Although cuserid() does not make use of getpwnam() in Release 4.1, the pointer returned by each points to the same static array. Data in this array may be overwritten by a subsequent call to either routine.

9.2.2.4 Errors

These routines depend on malloc(3) and fopen(2), either of which may fail and return an error.

POSIX.1 Section 10, Data Interchange Format

10.1 Archive/Interchange File Format

Release 4.1 provides a copying utility named pax(1), which reads and writes tar(1) and cpio(1) archives that conform to the standard. For backward compatibility, pax can also read, but not write, a number of other archive formats, such as UNIX Version 7 tar and System V binary cpio archives.

10.1.1 Extended tar Format

When an invalid filename is encountered, pax skips the file.

10.1.2 Extended cpio Format

10.1.2.1 Header

The value of c_dev is taken from the file system's device number. c_ino is taken from the file's inode number. c_rdev is taken from the device number of a special file.

10.1.2.2 File Name

When an invalid filename is encountered, pax skips the file.

10.1.2.4 Special Entries

c_filesize is zero for block special and character special files.

10.1.2.5 cpio Values

pax supports the permissions, file types, and mode masks. In the <sys/stat.h> header, Release 4.1 defines constants that are equivalent to those listed in the standard:

POSIX.1 File Permissions	***4.1 Equivalents***	Function
C_IRUSR	S_IRUSR	read permission, owner
C_IWUSR	S_IWUSR	write permission, owner
C_IXUSR	S_IXUSR	execute/search permission, owner
C_IRGRP	S_IRGRP	read permission, group
C_IWGRP	S_IWGRP	write permission, group
C_IXGRP	S_IXGRP	execute/search permission, group
C_IROTH	S_IROTH	read permission, other
C_IWOTH	S_IWOTH	write permission, other
C_IXOTH	S_IXOTH	execute/search permission, other
C_ISUID	S_ISUID	set user id on execution
C_ISGID	S_ISGID	set group id on execution
C_ISVTX	S_ISVTX	save swapped text even after use

POSIX.1 File Types	*4.1 Equivalents*	*Meaning*
C_ISDIR	S_IFDIR	directory
C_ISFIFO	S_IFIFO	FIFO
C_ISREG	S_IFREG	regular
C_ISBLK	S_IFBLK	block special
C_ISCHR	S_IFCHR	character special
C_ISCTG	S_IFCTG	*unused*
C_ISLNK	S_IFLNK	symbolic link
C_ISSOCK	S_IFSOCK	socket

pax ignores file modes other than file permissions.

10.1.3 Multiple Volumes

When pax encounters an end-of-file or end-of-medium condition, it issues a prompt so that the user may load the next volume, and waits for a response from standard input before proceeding.

Headers

The <limits.h> Header

```
/*      @(#)limits.h 1.11 89/06/16 SMI; from S5R2 1.1   */

#ifndef __sys_limits_h
#define __sys_limits_h

#define CHAR_BIT                 0x8
#define SCHAR_MIN               -0x80
#define SCHAR_MAX                0x7F
#define UCHAR_MAX                0xFF
#define CHAR_MIN                -0x80
#define CHAR_MAX                 0x7F
#define SHRT_MIN                -0x8000
#define SHRT_MAX                 0x7FFF
#define USHRT_MAX                0xFFFF
#define INT_MIN                 -0x80000000
#define INT_MAX                  0x7FFFFFFF
#define UINT_MAX                 0xFFFFFFFF
#define LONG_MIN                -0x80000000
#define LONG_MAX                 0x7FFFFFFF
#define ULONG_MAX                0xFFFFFFFF
#define MB_LEN_MAX              4

/*
 * All POSIX systems must support the following values
 * A system may support less restrictive values
 */
#define _POSIX_ARG_MAX          4096
#define _POSIX_CHILD_MAX        6
#define _POSIX_LINK_MAX         8
#define _POSIX_MAX_CANON        255
#define _POSIX_MAX_INPUT        255
#define _POSIX_NAME_MAX         14
#define _POSIX_NGROUPS_MAX      0
#define _POSIX_OPEN_MAX         16
#define _POSIX_PATH_MAX         255
#define _POSIX_PIPE_BUF         512

#endif  /* !__sys_limits_h */
```

The <unistd.h> Header

```
/*      @(#)unistd.h 1.8 89/06/25 SMI; from S5R3 1.5 */

#ifndef __sys_unistd_h
#define __sys_unistd_h

#define _SC_ARG_MAX             1       /* space for argv & envp */
#define _SC_CHILD_MAX           2       /* maximum children per process */
#define _SC_CLK_TCK             3       /* clock ticks/sec */
#define _SC_NGROUPS_MAX         4       /* number of groups if multple supp. */
#define _SC_OPEN_MAX            5       /* max open files per process */
#define _SC_JOB_CONTROL         6       /* do we have job control */
#define _SC_SAVED_IDS           7       /* do we have saved uid/gids */
#define _SC_VERSION             8       /* POSIX version supported */

#define _POSIX_JOB_CONTROL      1
#define _POSIX_SAVED_IDS        1
#define _POSIX_VERSION          198808

#define _PC_LINK_MAX            1       /* max links to file/dir */
#define _PC_MAX_CANON           2       /* max line length */
#define _PC_MAX_INPUT           3       /* max "packet" to a tty device */
#define _PC_NAME_MAX            4       /* max pathname component length */
#define _PC_PATH_MAX            5       /* max pathname length */
#define _PC_PIPE_BUF            6       /* size of a pipe */
#define _PC_CHOWN_RESTRICTED    7       /* can we give away files */
#define _PC_NO_TRUNC            8       /* trunc or error on >NAME_MAX */
#define _PC_VDISABLE            9       /* best char to shut off tty c_cc */
#define _PC_LAST                9       /* highest value of any _PC_* */

#define STDIN_FILENO    0
#define STDOUT_FILENO   1
#define STDERR_FILENO   2

#ifndef _POSIX_SOURCE
/*
 * SVID lockf() requests
 */
#define F_ULOCK         0       /* Unlock a previously locked region */
#define F_LOCK          1       /* Lock a region for exclusive use */
#define F_TLOCK         2       /* Test and lock a region for exclusive use */
#define F_TEST          3       /* Test a region for other processes locks */

#endif  !_POSIX_SOURCE

/*
 * lseek & access args
 *
 * SEEK_* have to track L_* in sys/file.h
 * ?_OK have to track ?_OK in sys/file.h
 */
#define SEEK_SET        0       /* Set file pointer to "offset" */
#define SEEK_CUR        1       /* Set file pointer to current plus "offset" */
#define SEEK_END        2       /* Set file pointer to EOF plus "offset" */

#define F_OK            0       /* does file exist */
#define X_OK            1       /* is it executable by caller */
#define W_OK            2       /* is it writable by caller */
#define R_OK            4       /* is it readable by caller */

#if     !defined(KERNEL)
```

```
#include <sys/types.h>

extern void     _exit(/* int status */);
extern int      access(/* char *path, int amode */);
extern unsigned alarm(/* unsigned secs */);
extern int      chdir(/* char *path */);
extern int      chmod(/* char *path, mode_t mode */);
extern int      chown(/* char *path, uid_t owner, gid_t group */);
extern int      close(/* int fildes */);
extern char     *ctermid(/* char *s */);
extern char     *cuserid(/* char *s */);
extern int      dup(/* int fildes */);
extern int      dup2(/* int fildes, int fildes2 */);
extern int      execl(/* char *path, ... */);
extern int      execle(/* char *path, ... */);
extern int      execlp(/* char *file, ... */);
extern int      execv(/* char *path, char *argv[] */);
extern int      execve(/* char *path, char *argv[], char *envp[] */);
extern int      execvp(/* char *file, char *argv[] */);
extern pid_t    fork(/* void */);
extern long     fpathconf(/* int fd, int name */);
extern char     *getcwd(/* char *buf, int size */);
extern gid_t    getegid(/* void */);
extern uid_t    geteuid(/* void */);
extern gid_t    getgid(/* void */);
extern int      getgroups(/* int gidsetsize, gid_t grouplist[] */);
extern char     *getlogin(/* void */);
extern pid_t    getpgrp(/* void */);
extern pid_t    getpid(/* void */);
extern pid_t    getppid(/* void */);
extern uid_t    getuid(/* void */);
extern int      isatty(/* int fildes */);
extern int      link(/* char *path1, char *path2 */);
extern off_t    lseek(/* int fildes, off_t offset, int whence */);
extern long     pathconf(/* char *path, int name */);
extern int      pause(/* void */);
extern int      pipe(/* int fildes[2] */);
extern int      read(/* int fildes, char *buf, unsigned int nbyte */);
extern int      rmdir(/* char *path */);
extern int      setgid(/* gid_t gid */);
extern int      setpgid(/* pid_t pid, pid_t pgid */);
extern pid_t    setsid(/* void */);
extern int      setuid(/* uid_t uid */);
extern unsigned sleep(/* unsigned int seconds */);
extern long     sysconf(/* int name */);
extern pid_t    tcgetpgrp(/* int fildes */);
extern int      tcsetpgrp(/* int fildes, pid_t pgrp_id */);
extern char     *ttyname(/* int fildes */);
extern int      unlink(/* char *path */);
extern int      write(/* int fildes, char *buf, unsigned int nbyte */);

#endif  /* !KERNEL */
#endif  /* !__sys_unistd_h */
```

A

ISO Latin 1 Character Set

The following table displays the ISO 8859/1 character set.

Table A-1 *ISO Latin 1*

Row/Col	Decimal	Octal		Name
02/00	032	040	SP	SPACE
02/01	033	041	!	EXCLAMATION POINT
02/02	034	042	"	QUOTATION MARK
02/03	035	043	#	NUMBER SIGN
02/04	036	044	$	DOLLAR SIGN
02/05	037	045	%	PERCENT SIGN
02/06	038	046	&	AMPERSAND
02/07	039	047	'	APOSTROPHE
02/08	040	050	(	LEFT PARENTHESIS
02/09	041	051	)	RIGHT PARENTHESIS
02/10	042	052	*	ASTERISK
02/11	043	053	+	PLUS SIGN
02/12	044	054	,	COMMA
02/13	045	055	-	HYPHEN, MINUS SIGN
02/14	046	056	.	FULL STOP (U.S.: PERIOD, DECIMAL POINT)
02/15	047	057	/	SOLIDUS (U.S.: SLASH)
03/00	048	060	0	DIGIT ZERO
03/01	049	061	1	DIGIT ONE
03/02	050	062	2	DIGIT TWO
03/03	051	063	3	DIGIT THREE
03/04	052	064	4	DIGIT FOUR
03/05	053	065	5	DIGIT FIVE
03/06	054	066	6	DIGIT SIX
03/07	055	067	7	DIGIT SEVEN
03/08	056	070	8	DIGIT EIGHT
03/09	057	071	9	DIGIT NINE
03/10	058	072	:	COLON
03/11	059	073	;	SEMICOLON
03/12	060	074	<	LESS-THAN SIGN
03/13	061	075	=	EQUALS SIGN
03/14	062	076	>	GREATER-THAN SIGN
03/15	063	077	?	QUESTION MARK
04/00	064	100	@	COMMERCIAL AT
04/01	065	101	A	LATIN CAPITAL LETTER A
04/02	066	102	B	LATIN CAPITAL LETTER B
04/03	067	103	C	LATIN CAPITAL LETTER C
04/04	068	104	D	LATIN CAPITAL LETTER D

Table A-1 *ISO Latin 1— Continued*

Row/Col	Decimal	Octal		Name
04/05	069	105	E	LATIN CAPITAL LETTER E
04/06	070	106	F	LATIN CAPITAL LETTER F
04/07	071	107	G	LATIN CAPITAL LETTER G
04/08	072	110	H	LATIN CAPITAL LETTER H
04/09	073	111	I	LATIN CAPITAL LETTER I
04/10	074	112	J	LATIN CAPITAL LETTER J
04/11	075	113	K	LATIN CAPITAL LETTER K
04/12	076	114	L	LATIN CAPITAL LETTER L
04/13	077	115	M	LATIN CAPITAL LETTER M
04/14	078	116	N	LATIN CAPITAL LETTER N
04/15	079	117	O	LATIN CAPITAL LETTER O
05/00	080	120	P	LATIN CAPITAL LETTER P
05/01	081	121	Q	LATIN CAPITAL LETTER Q
05/02	082	122	R	LATIN CAPITAL LETTER R
05/03	083	123	S	LATIN CAPITAL LETTER S
05/04	084	124	T	LATIN CAPITAL LETTER T
05/05	085	125	U	LATIN CAPITAL LETTER U
05/06	086	126	V	LATIN CAPITAL LETTER V
05/07	087	127	W	LATIN CAPITAL LETTER W
05/08	088	130	X	LATIN CAPITAL LETTER X
05/09	089	131	Y	LATIN CAPITAL LETTER Y
05/10	090	132	Z	LATIN CAPITAL LETTER Z
05/11	091	133	[	LEFT SQUARE BRACKET
05/12	092	134	\	REVERSE SOLIDUS (U.S.: BACK SLASH)
05/13	093	135	]	RIGHT SQUARE BRACKET
05/14	094	136	^	CIRCUMFLEX ACCENT
05/15	095	137	_	LOW LINE (U.S.: UNDERSCORE)
06/00	096	140	‘	GRAVE ACCENT
06/01	097	141	a	LATIN SMALL LETTER a
06/02	098	142	b	LATIN SMALL LETTER b
06/03	099	143	c	LATIN SMALL LETTER c
06/04	100	144	d	LATIN SMALL LETTER d
06/05	101	145	e	LATIN SMALL LETTER e
06/06	102	146	f	LATIN SMALL LETTER f
06/07	103	147	g	LATIN SMALL LETTER g
06/08	104	150	h	LATIN SMALL LETTER h
06/09	105	151	i	LATIN SMALL LETTER i
06/10	106	152	j	LATIN SMALL LETTER j
06/11	107	153	k	LATIN SMALL LETTER k
06/12	108	154	l	LATIN SMALL LETTER l
06/13	109	155	m	LATIN SMALL LETTER m
06/14	110	156	n	LATIN SMALL LETTER n
06/15	111	157	o	LATIN SMALL LETTER o
07/00	112	160	p	LATIN SMALL LETTER p
07/01	113	161	q	LATIN SMALL LETTER q
07/02	114	162	r	LATIN SMALL LETTER r
07/03	115	163	s	LATIN SMALL LETTER s
07/04	116	164	t	LATIN SMALL LETTER t
07/05	117	165	u	LATIN SMALL LETTER u
07/06	118	166	v	LATIN SMALL LETTER v
07/07	119	167	w	LATIN SMALL LETTER w
07/08	120	170	x	LATIN SMALL LETTER x
07/09	121	171	y	LATIN SMALL LETTER y

Table A-1 *ISO Latin 1— Continued*

Row/Col	Decimal	Octal		Name
07/10	122	172	z	LATIN SMALL LETTER z
07/11	123	173	{	LEFT CURLY BRACKET
07/12	124	174	|	VERTICAL LINE
07/13	125	175	}	RIGHT CURLY BRACKET
07/14	126	176	~	TILDE
10/00	160	240		NO-BREAK SPACE
10/01	161	241		INVERTED EXCLAMATION MARK
10/02	162	242		CENT SIGN
10/03	163	243		POUND SIGN
10/04	164	244		CURRENCY SIGN
10/05	165	245		YEN SIGN
10/06	166	246		BROKEN BAR
10/07	167	247		PARAGRAPH SIGN, (U.S.: SECTION SIGN)
10/08	168	250		DIAERESIS
10/09	169	251		COPYRIGHT SIGN
10/10	170	252		FEMININE ORDINAL INDICATOR
10/11	171	253		LEFT ANGLE QUOTATION MARK
10/12	172	254		NOT SIGN
10/13	173	255		SHY SOFT HYPHEN
10/14	174	256		REGISTERED TRADEMARK SIGN
10/15	175	257		MACRON
11/00	176	260		RING ABOVE, DEGREE SIGN
11/01	177	261		PLUS-MINUS SIGN
11/02	178	262		SUPERSCRIPT TWO
11/03	179	263		SUPERSCRIPT THREE
11/04	180	264		ACUTE ACCENT
11/05	181	265		MICRO SIGN
11/06	182	266		PILCROW SIGN, (U.S.: PARAGRAPH)
11/07	183	267		MIDDLE DOT
11/08	184	270		CEDILLA
11/09	185	271		SUPERSCRIPT ONE
11/10	186	272		MASCULINE ORDINAL INDICATOR
11/11	187	273		RIGHT ANGLE QUOTATION MARK
11/12	188	274		VULGAR FRACTION ONE QUARTER
11/13	189	275		VULGAR FRACTION ONE HALF
11/14	190	276		VULGAR FRACTION THREE QUARTERS
11/15	191	277		INVERTED QUESTION MARK
12/00	192	300		LATIN CAPITAL LETTER A WITH GRAVE ACCENT
12/01	193	301		LATIN CAPITAL LETTER A WITH ACUTE ACCENT
12/02	194	302		LATIN CAPITAL LETTER A WITH CIRCUMFLEX ACCENT
12/03	195	303		LATIN CAPITAL LETTER A WITH TILDE
12/04	196	304		LATIN CAPITAL LETTER A WITH DIAERESIS
12/05	197	305		LATIN CAPITAL LETTER A WITH RING ABOVE
12/06	198	306		CAPITAL DIPHTHONG AE
12/07	199	307		LATIN CAPITAL LETTER C WITH CEDILLA
12/08	200	310		LATIN CAPITAL LETTER E WITH GRAVE ACCENT
12/09	201	311		LATIN CAPITAL LETTER E WITH ACUTE ACCENT
12/10	202	312		LATIN CAPITAL LETTER E WITH CIRCUMFLEX ACCENT
12/11	203	313		LATIN CAPITAL LETTER E WITH DIAERESIS
12/12	204	314		LATIN CAPITAL LETTER I WITH GRAVE ACCENT
12/13	205	315		LATIN CAPITAL LETTER I WITH ACUTE ACCENT
12/14	206	316		LATIN CAPITAL LETTER I WITH CIRCUMFLEX ACCENT
12/15	207	317		LATIN CAPITAL LETTER I WITH DIAERESIS

Table A-1 *ISO Latin 1— Continued*

Row/Col	Decimal	Octal		Name
13/00	208	320		CAPITAL ICELANDIC LETTER ETH
13/01	209	321		LATIN CAPITAL LETTER N WITH TILDE
13/02	210	322		LATIN CAPITAL LETTER O WITH GRAVE ACCENT
13/03	211	323		LATIN CAPITAL LETTER O WITH ACUTE ACCENT
13/04	212	324		LATIN CAPITAL LETTER O WITH CIRCUMFLEX ACCENT
13/05	213	325		LATIN CAPITAL LETTER O WITH TILDE
13/06	214	326		LATIN CAPITAL LETTER O WITH DIAERESIS
13/07	215	327		MULTIPLICATION SIGN
13/08	216	330		LATIN CAPITAL LETTER O WITH OBLIQUE STROKE
13/09	217	331		LATIN CAPITAL LETTER U WITH GRAVE ACCENT
13/10	218	332		LATIN CAPITAL LETTER U WITH ACUTE ACCENT
13/11	219	333		LATIN CAPITAL LETTER U WITH CIRCUMFLEX
13/12	220	334		LATIN CAPITAL LETTER U WITH DIAERESIS
13/13	221	335		LATIN CAPITAL LETTER Y WITH ACUTE ACCENT
13/14	222	336		CAPITAL ICELANDIC LETTER THORN
13/15	223	337		SMALL GERMAN LETTER SHARP s
14/00	224	340		LATIN SMALL LETTER a WITH GRAVE ACCENT
14/01	225	341		LATIN SMALL LETTER a WITH ACUTE ACCENT
14/02	226	342		LATIN SMALL LETTER a WITH CIRCUMFLEX ACCENT
14/03	227	343		LATIN SMALL LETTER a WITH TILDE
14/04	228	344		LATIN SMALL LETTER a WITH DIAERESIS
14/05	229	345		LATIN SMALL LETTER a WITH RING ABOVE
14/06	230	346		SMALL DIPHTHONG ae
14/07	231	347		LATIN SMALL LETTER c WITH CEDILLA
14/08	232	350		LATIN SMALL LETTER e WITH GRAVE ACCENT
14/09	233	351		LATIN SMALL LETTER e WITH ACUTE ACCENT
14/10	234	352		LATIN SMALL LETTER e WITH CIRCUMFLEX ACCENT
14/11	235	353		LATIN SMALL LETTER e WITH DIAERESIS
14/12	236	354		LATIN SMALL LETTER i WITH GRAVE ACCENT
14/13	237	355		LATIN SMALL LETTER i WITH ACUTE ACCENT
14/14	238	356		LATIN SMALL LETTER i WITH CIRCUMFLEX ACCENT
14/15	239	357		LATIN SMALL LETTER i WITH DIAERESIS
15/00	240	360		SMALL ICELANDIC LETTER ETH
15/01	241	361		LATIN SMALL LETTER n WITH TILDE
15/02	242	362		LATIN SMALL LETTER o WITH GRAVE ACCENT
15/03	243	363		LATIN SMALL LETTER o WITH ACUTE ACCENT
15/04	244	364		LATIN SMALL LETTER o WITH CIRCUMFLEX ACCENT
15/05	245	365		LATIN SMALL LETTER o WITH TILDE
15/06	246	366		LATIN SMALL LETTER o WITH DIAERESIS
15/07	247	367		DIVISION SIGN
15/08	248	370		LATIN SMALL LETTER o WITH OBLIQUE STROKE
15/09	249	371		LATIN SMALL LETTER u WITH GRAVE ACCENT
15/10	250	372		LATIN SMALL LETTER u WITH ACUTE ACCENT
15/11	251	373		LATIN SMALL LETTER u WITH CIRCUMFLEX ACCENT
15/12	252	374		LATIN SMALL LETTER u WITH DIAERESIS
15/13	253	375		LATIN SMALL LETTER y WITH ACUTE ACCENT
15/14	254	376		SMALL ICELANDIC LETTER THORN
15/15	255	377		LATIN SMALL LETTER y WITH DIAERESIS

Table A-2 *The ISO 8859 Standard Character Set Family*

No	Name	Coverage	Status	Release 4.1 Support
1	Latin Alphabet #1	Western European	Approved Intl Standard	Supported
2	Latin Alphabet #2	Eastern European	Approved Intl Standard	Not Supported
3	Latin Alphabet #3	Southern European and Southern Africa	Approved Intl Standard	Not Supported
4	Latin Alphabet #4	Majority of Scandanavian C's	Approved Intl Standard	Not Supported
5	Latin-Cyrillic Alphabet	ASCII + Cyrillic	Approved-Not Published	Not Supported
6	Latin-Arabic Alphabet	ASCII + Arabic	Approved Intl Standard	Not Supported
7	Latin-Greek Alphabet	ASCII + Greek	Approved Intl Standard	Not Supported
8	Latin-Hebrew Alphabet	ASCII + Hebrew	Approved-Not Published	Not Supported
9	Latin Alphabet #5	Turkish Changes to 8859/3	Proposed	Not Supported

B

U.S. and European Keyboard Layouts

Figure B-1 *United States*

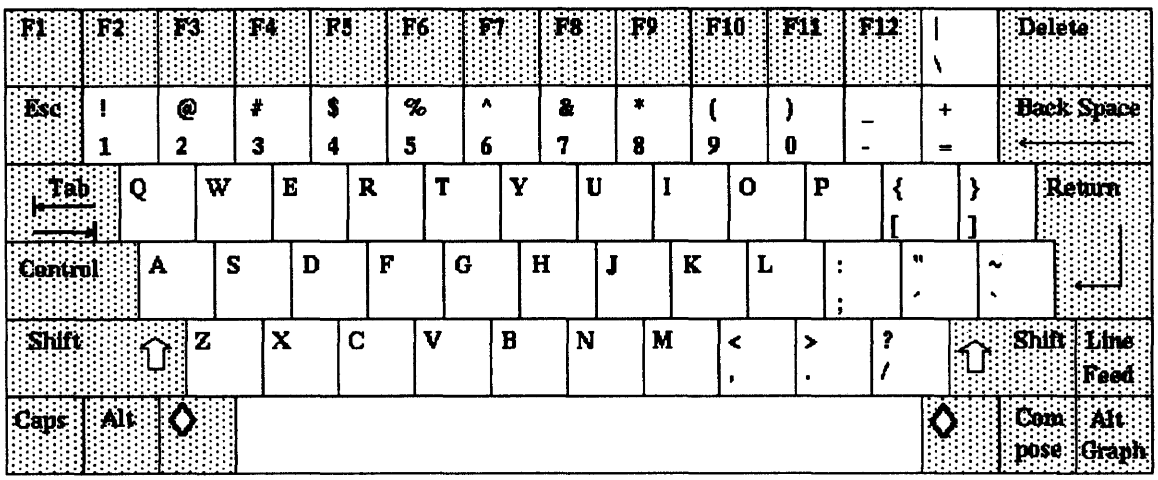


Figure B-2 *Belgium/France*

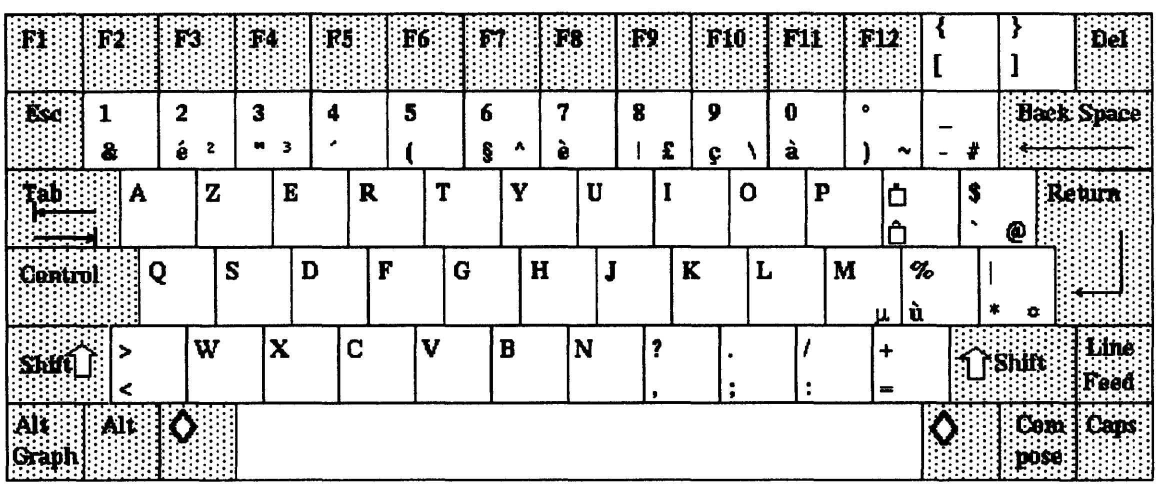


Figure B-3 *Canada*

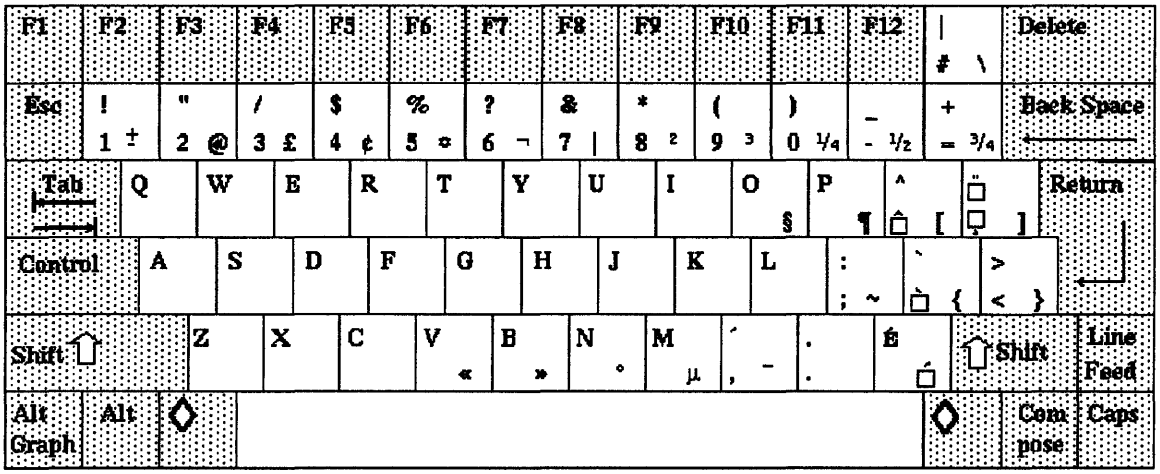


Figure B-4 *Denmark*

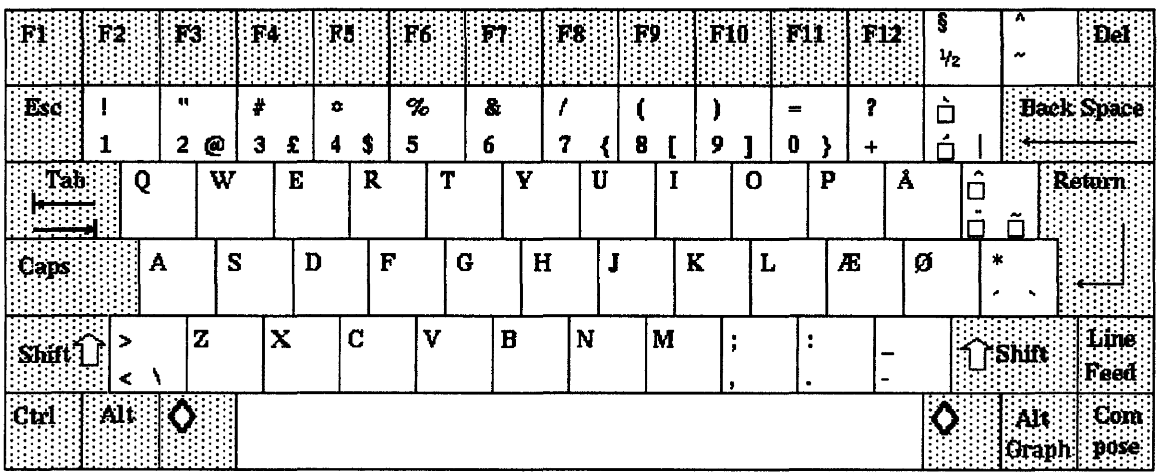


Figure B-5 *Netherlands*

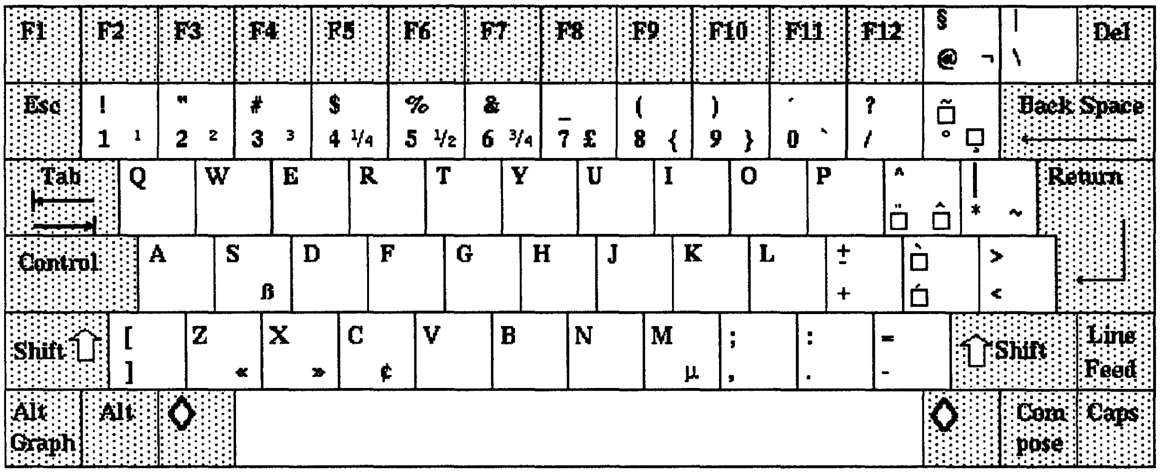


Figure B-6 *Germany*

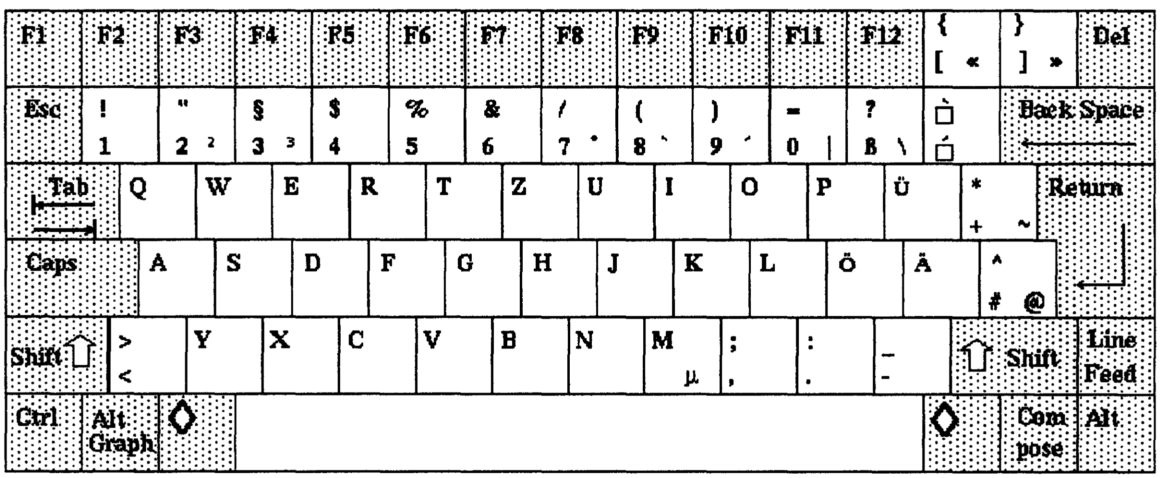


Figure B-7 *Italy*

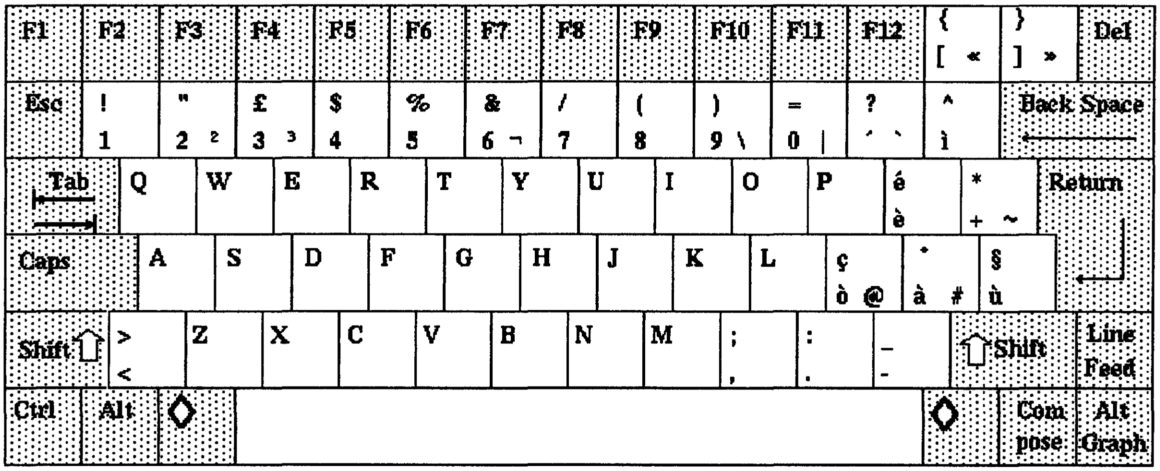


Figure B-8 *Norway*

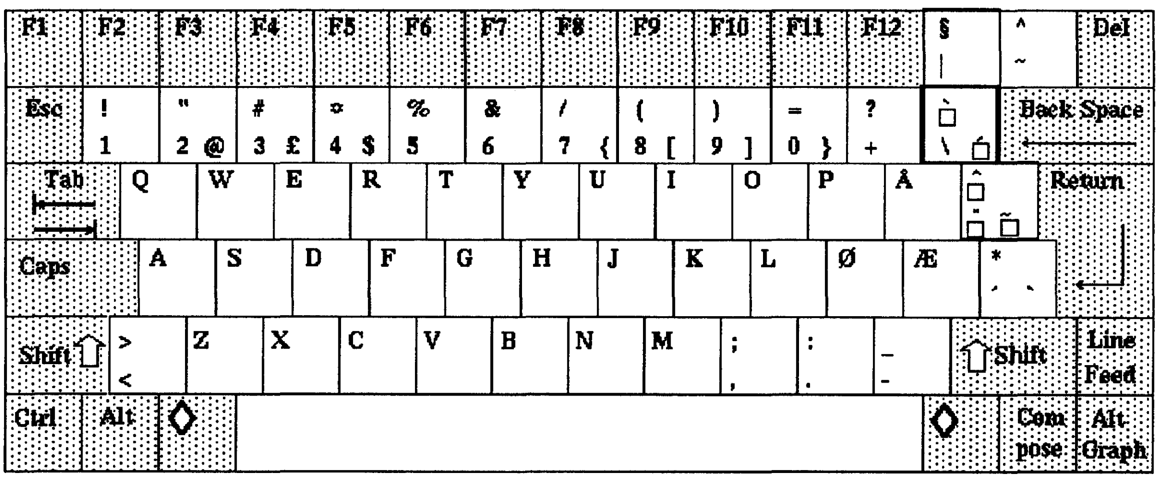


Figure B-9 *Portugal*

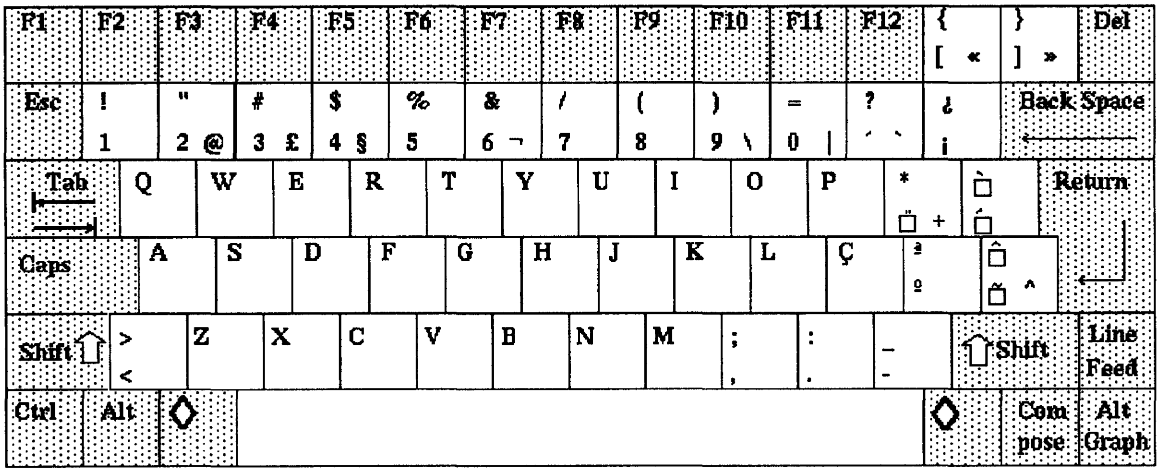


Figure B-10 *Spain*

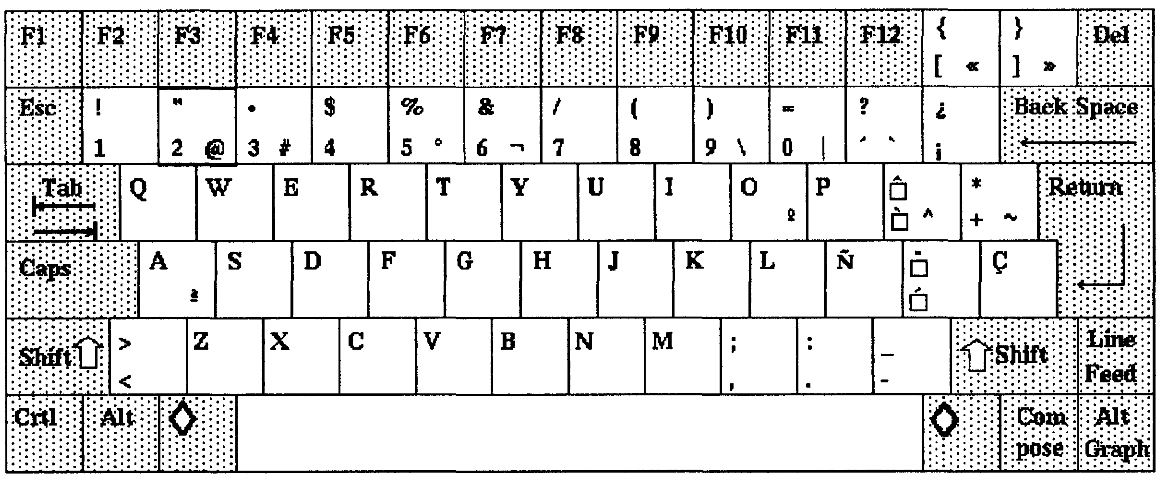


Figure B-11 *Sweden/Finland*

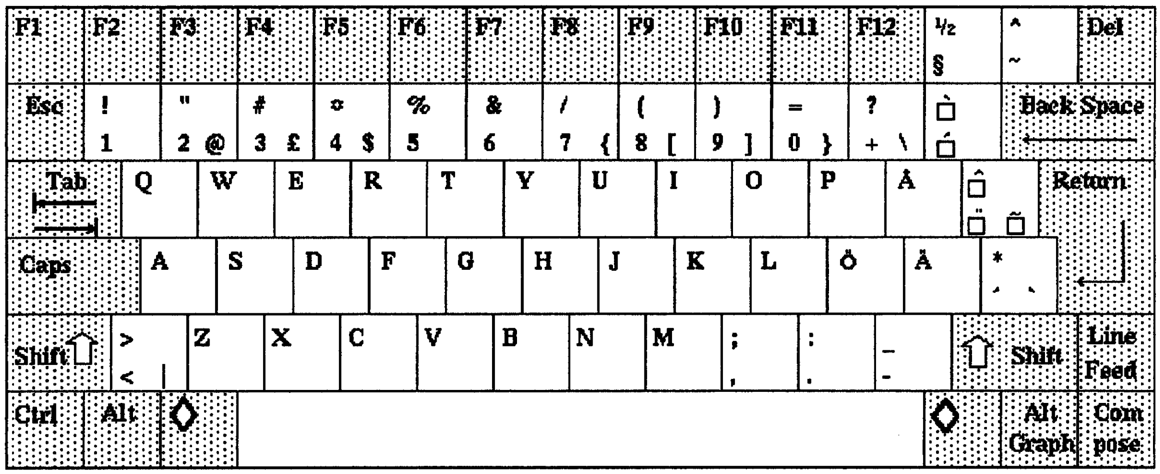


Figure B-12 *Switzerland (French)*

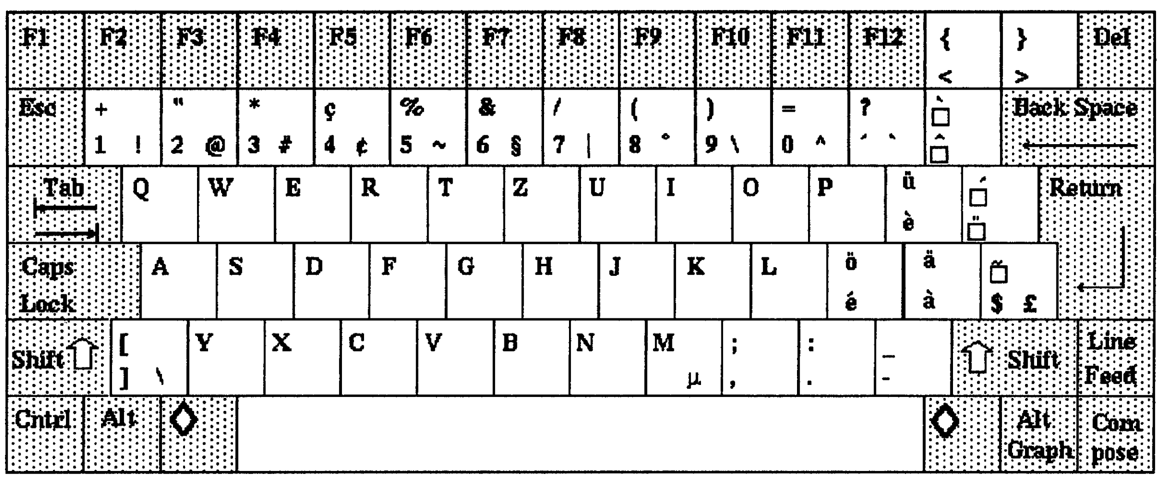


Figure B-13 *Switzerland (German)*

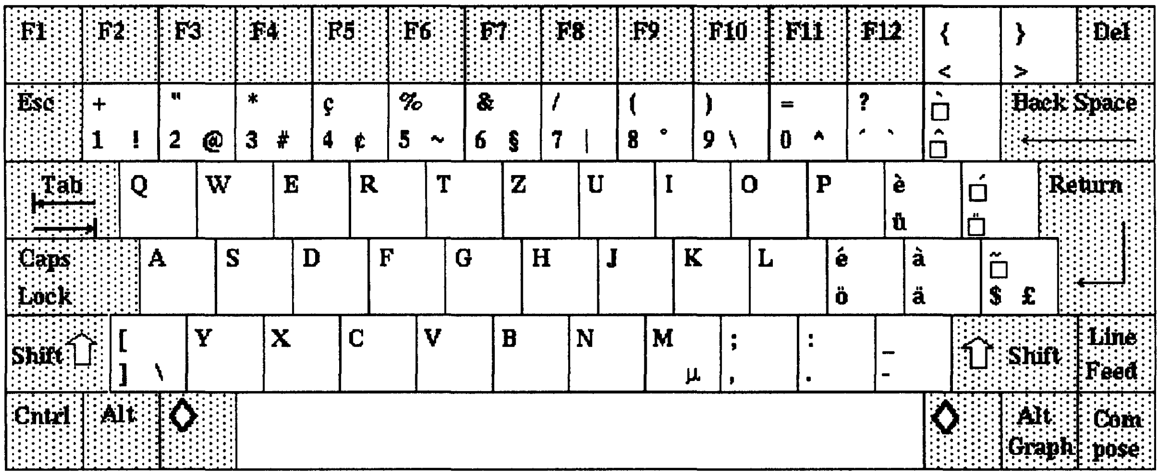
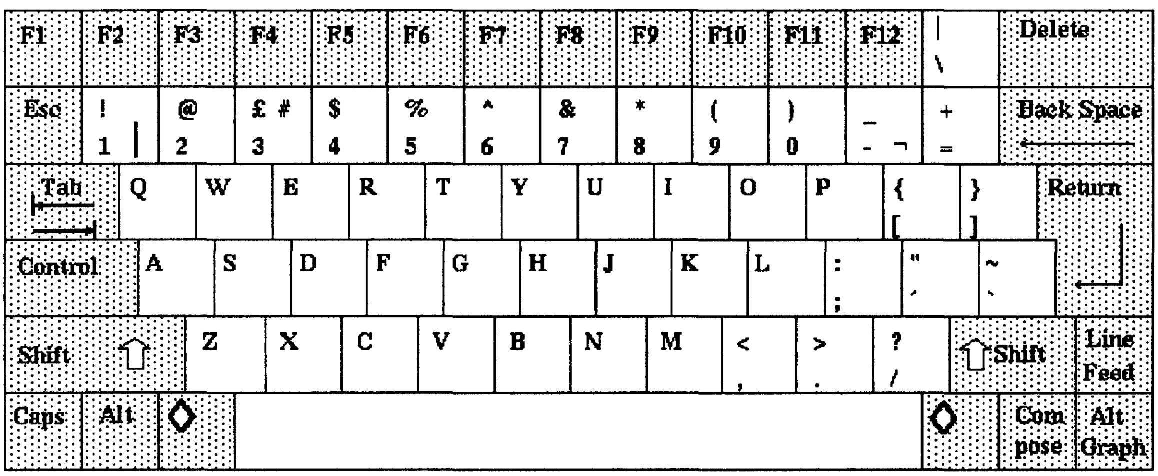


Figure B-14 *United Kingdom*



C

Compose Key and Floating Accent Key Sequences

Table C-1 *Compose Key Sequences*

Compose Key Character Sequences			
Key	Key	ISO Latin 1 Code	Description
Space	Space	0xA0	non-overstrike backspace
!	!	0xA1	inverted !
c	/	0xA2	cent sign
C	/	0xA2	cent sign
l	-	0xA3	Pounds Sterling
L	-	0xA3	Pounds Sterling
o	x	0xA4	currency symbol
O	X	0xA4	currency symbol
0	x	0xA4	currency symbol
0	X	0xA4	currency symbol
Y	-	0xA5	Yen
y	-	0xA5	Yen
|	|	0xA6	broken bar
s	o	0xA7	section mark
S	O	0xA7	section mark
"	"	0xA8	diaeresis
c	o	0xA9	copyright
C	O	0xA9	copyright
-	a	0xAA	feminine superior numeral
-	A	0xAA	feminine superior numeral
<	<	0xAB	left guillemot
-	|	0xAC	not sign
-	,	0xAC	not sign
-	-	0xAD	soft hyphen
r	o	0xAE	registered
R	O	0xAE	registered
^	-	0xAF	macron (?)
^	*	0xB0	degree
0	^	0xB0	degree
+	-	0xB1	plus/minus
^	2	0xB2	superior '2'
^	3	0xB3	superior '3'

Table C-1 *Compose Key Sequences— Continued*

Compose Key Character Sequences			
Key	Key	ISO Latin 1 Code	Description
\	\	0xB4	acute accent
/	u	0xB5	mu
P	!	0xB6	paragraph mark
^	.	0xB7	centered dot
,	,	0xB8	cedilla
^	1	0xB9	superior '1'
_	o	0xBA	masculine superior numeral
_	O	0xBA	masculine superior numeral
>	>	0xBB	right guillemot
1	4	0xBC	1/4
1	2	0xBD	1/2
3	4	0xBE	3/4
?	?	0xBF	inverted ?
A	`	0xC0	A with grave accent
A	'	0xC1	A with acute accent
A	^	0xC2	A with circumflex accent
A	~	0xC3	A with tilde
A	"	0xC4	A with diaeresis
A	*	0xC5	A with ring
A	E	0xC6	AE dipthong
C	,	0xC7	C with cedilla
E	`	0xC8	E with grave accent
E	'	0xC9	E with acute accent
E	^	0xCA	E with circumflex accent
E	"	0xCB	E with diaeresis
I	`	0xCC	I with grave accent
I	'	0xCD	I with acute accent
I	^	0xCE	I with circumflex accent
I	"	0xCF	I with diaeresis
D	-	0xD0	Upper-case eth(?)
N	~	0xD1	N with tilde
O	`	0xD2	O with grave accent
O	'	0xD3	O with acute accent
O	^	0xD4	O with circumflex accent
O	~	0xD5	O with tilde
O	"	0xD6	O with diaeresis
x	x	0xD7	multiplication sign
O	/	0xD8	O with slash
U	`	0xD9	U with grave accent
U	'	0xDA	U with acute accent
U	^	0xDB	U with circumflex accent
U	"	0xDC	U with diaeresis
Y	'	0xDD	Y with acute accent
P	|	0xDE	Upper-case thorn

Table C-1 *Compose Key Sequences— Continued*

Compose Key Character Sequences			
Key	Key	ISO Latin 1 Code	Description
T	H	0xDE	Upper-case thorn
s	s	0xDF	German double-s
a	\`	0xE0	a with grave accent
a	'	0xE1	a with acute accent
a	^	0xE2	a with circumflex accent
a	~	0xE3	a with tilde
a	"	0xE4	a with diaeresis
a	*	0xE5	a with ring
a	e	0xE6	ae dipthong
c	,	0xE7	c with cedilla
e	\`	0xE8	e with grave accent
e	'	0xE9	e with acute accent
e	^	0xEA	e with circumflex accent
e	"	0xEB	e with diaeresis
i	\`	0xEC	i with grave accent
i	'	0xED	i with acute accent
i	^	0xEE	i with circumflex accent
i	"	0xEF	i with diaeresis
d	-	0xF0	Lower-case eth(?)
n	~	0xF1	n with tilde
o	\`	0xF2	o with grave accent
o	'	0xF3	o with acute accent
o	^	0xF4	o with circumflex accent
o	~	0xF5	o with tilde
o	"	0xF6	o with diaeresis
-	:	0xF7	division sign
o	/	0xF8	o with slash
u	\`	0xF9	u with grave accent
u	'	0xFA	u with acute accent
u	^	0xFB	u with circumflex accent
u	"	0xFC	u with diaeresis
y	'	0xFD	y with acute accent
p	|	0xFE	Lower-case thorn
t	h	0xFE	Lower-case thorn
y	"	0xFF	y with diaeresis

Table C-2 *Floating Accent Key Sequences*

Floating Accent Key Character Sequences			
Key	Key	ISO Latin 1 Code	Description
Umlaut	A	0xC4	A with umlaut
	E	0xCB	E with umlaut
	I	0xCF	I with umlaut
	O	0xD6	O with umlaut
	U	0xDC	U with umlaut
	a	0xE4	a with umlaut
	e	0xEB	e with umlaut
	i	0xEF	i with umlaut
	o	0xF6	o with umlaut
	u	0xFC	u with umlaut
	y	0xFF	y with umlaut
Circumflex	A	0xC2	A with circumflex
	E	0xCA	E with circumflex
	I	0xCE	I with circumflex
	O	0xD4	O with circumflex
	U	0xDB	U with circumflex
	a	0xE2	a with circumflex
	e	0xEA	e with circumflex
	i	0xEE	i with circumflex
	o	0xF4	o with circumflex
	u	0xFB	u with circumflex
Tilde	A	0xC3	A with tilde
	N	0xD1	N with tilde
	O	0xD5	O with tilde
	a	0xE3	a with tilde
	n	0xF1	n with tilde
	o	0xF5	o with tilde
Cedilla	C	0xC7	C with cedilla
	c	0xE7	c with cedilla
Acute Accent	A	0xC1	A with acute accent
	E	0xC9	E with acute accent
	I	0xCD	I with acute accent
	O	0xD3	O with acute accent
	U	0xDA	U with acute accent
	a	0xE1	a with acute accent
	e	0xE9	e with acute accent
	i	0xED	i with acute accent
	o	0xF3	o with acute accent
	u	0xFA	u with acute accent
	y	0xFD	y with acute accent

Table C-2 *Floating Accent Key Sequences— Continued*

Floating Accent Key Character Sequences			
Key	Key	ISO Latin 1 Code	Description
Grave Accent	A	0xC0	A with grave accent
	E	0xC8	E with grave accent
	I	0xCC	I with grave accent
	O	0xD2	O with grave accent
	U	0xD9	U with grave accent
	a	0xE0	a with grave accent
	e	0xE8	e with grave accent
	i	0xEC	i with grave accent
	o	0xF2	o with grave accent
	u	0xF9	u with grave accent

Index